



Developer Reference Manual

Commodore-Amiga, Inc.

Commodore-Amiga, Inc.

West Chester, Pennsylvania

Authors:

Dan Baker, Peter Cherna, Eric Cotton, Andy Finkel, Darren Greenwald, Jim Hawkins, Paul Higginbottom, Mike Kawahara, Perry Kivolowitz, Scott Lamb, Adam Levin-Delson, Chris Ludwig, Bryce Nesbitt, Benjamin Phister, Mark Ricci, Stephen D. Ritchie, David Rosen, Carl Sassenrath, Carolyn Scheppner, Leo L. Schwab, Roy Strauss, Guy Wright, Ken Yeast

Contributors:

Dan Baker, John Campbell, Louise Carroll, Jerry Hartzler, E.J. Mungin, John Orr, Sherrie Rubincan, Gail Wellington

Special thanks to Perry Kivolowitz of ASDG, Inc. and Scott Lamb of Merit Software, Inc.

Editor:

Mark Ricci

Copyright © 1992 by Commodore-Electronics, Ltd.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Commodore-Amiga was aware of a trademark claim, the designations have been printed in initial caps. CDTV is a registered trademark of Commodore Electronics Limited. Amiga is a registered trademark of Commodore-Amiga, Inc. Amiga 500, Amiga 1000, Amiga 2000, Amiga 3000, AmigaDOS, Amiga Workbench, and Amiga Kickstart are trademarks of Commodore-Amiga, Inc. AUTOCONFIG is a trademark of Commodore Electronics Limited. Commodore and the Commodore logo are registered trademarks of Commodore Electronics Limited. Motorola is a registered trademark and 68000, 68010, 68020, 68030, and 68040 are trademarks of Motorola, Inc. Macintosh is a registered trademark of Apple Computer, Inc. MS-DOS and Windows are registered trademarks of Microsoft, Inc.

First printing, March 1992

WARNING: The information described in this manual may contain errors or bugs, and may not function as described. All information is subject to enhancement or upgrade for any reason and without notice, including to fix bugs, add features, or change performance. As with all software upgrades, full compatibility, although a goal, cannot be guaranteed, and is in fact unlikely.

With this document Commodore makes no warranties or representations, either express, or implied, with respect to the products being supplied on an "AS IS" basis and is expressly subject to change without notice. The entire risk as to the use of this information is assumed by the user. IN NO EVENT WILL COMMODORE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY CLAIM ARISING OUT OF THE INFORMATION PRESENTED HEREIN, EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITIES OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY.

CONTENTS

1 INTRODUCTION AND PHILOSOPHY

1.1	Introducing CDTV Multimedia	1
-----	-----------------------------------	---

2 THE USER INTERFACE

2.1	User Interface Issues	
2.1.1	User Interface Guidelines	1
2.1.2	Designing Screens for CDTV Multimedia	41
2.1.3	Localizing CDTV and CDTV Applications	51
2.2	CDTV Title Issues	
2.2.1	CDTV Title Guidelines	55

3 PROGRAMMING AND CDTV MULTIMEDIA

3.1	Getting Started	
3.1.1	Programming CDTV	1
3.1.2	Recommended CDTV Development Environments	7
3.1.3	Developer's Introduction	13
3.1.4	General Amiga Development Guidelines	21
3.1.5	2.0 Compatibility Problem Areas	27
3.2	CDTV Specifics	
3.2.1	CDTV Feature Overview	41
3.2.2	Bootting A CDTV Application	45
3.2.3	Localization Programming	57
3.2.4	CDTV File System	63
3.2.5	CDTV Device	69
3.2.6	Bookmark and Cardmark Device Drivers	101
3.2.7	CDTV Printer Preferences	133
3.2.8	The Power of CDXL	141
3.2.9	CDXL Toolkit	147
3.2.10	CDXL Toolkit and Video Capture for CDTV	161
3.2.11	License Material Overview	173
3.3	Graphics	
3.3.1	CDTV Graphics	177

3.3.2	Getting The Best Image For Your CDTV Application	181
3.3.3	PAL/NTSC Issues	191
3.4	Sound	
3.4.1	CDTV Sound	197
3.4.2	8SVX: Playing Samples Larger Than 128K	205
3.4.3	Producing High Quality Digitized Multilingual Narrative Audio	213
3.4.4	CDTV Audio Cookbook	221
3.4.5	CD-DA Sound	231
3.4.6	MIDI	241
3.5	Creating CDTV Applications	
3.5.1	CTrac Emulation System	249
3.5.2	Speeding Up Your Titles	255
3.5.3	Optimal Disc Layout	267
3.6	Manufacturing CDTV Discs	
3.6.1	Pre-Mastering and Mastering for CDTV	271
3.7	Debugging CDTV Applications	
3.7.1	Debugging CDTV Software	283
3.7.2	Troubleshooting Your Software	299

A OVERVIEW OF CDTV AND A500 DIFFERENCES

B DISC PACKAGING STANDARDS AND GRAPHICS STANDARDS

C RELATED AMIGA ARTICLES

Modification for Switchable PAL/NTSC CDTV	1
International Keyboard Input	3
Finding the Aspect Ratio	5

D CDTV TECHNICAL SPECIFICATIONS

E ADMINISTRATIVE FORMS

F RESOURCES

Introducing CDTV Multimedia

Commodore has undertaken a bold initiative—to bring CD-ROM to the consumer market. Up to now, CD-ROM has been defined as a mass data storage medium for electronic publishing accessed through a disc drive peripheral to a desk-top computer. With CDTV®, Commodore launches a new consumer electronics category, Interactive Multimedia. Commodore's goal is to transform CD-ROM from a somewhat staid professional medium into a mass, popular medium analogous to trade book publishing, home video, video games and prerecorded audio.

In order to make CD-ROM a mass publishing medium, the computer must become a truly user friendly family product. It is estimated that 75 percent of U.S. households own at least one VCR while only about 25 percent have a home computer. Why? Or perhaps a better question: why has the computer remained a school or workplace-experienced product?

Through extensive research in the U.S. and Europe it has become clear to Commodore (as well as many other companies) that consumers—including many using computers at their workplaces—do not consider the computer a fun product. Rather, it is often considered intimidating, surely not designed for the family or living room. To try to overcome this deep-seated consumer resistance, Commodore has adopted a Trojan horse strategy: the computer has been reconceived and repositioned into a multifunctional home appliance.

The CDTV player integrates an Amiga 500 computer motherboard (built on a Motorola 68000 chip) with an ISO-9660 compatible CD-ROM disc drive within a sleek, black box, a consumer-familiar form factor resembling a conventional CD-audio player or VCR. Clearly, it is designed to be part of the living or family room, easily integrated into a household entertainment rack system. With four video-out ports, the player hooks up to any standard TV set (i.e., composite, S-Video or RF modulated) or RGB monitor. Its two RCA-type audio ports ensure compatibility with all home stereos; the player plays all CD-audio titles as well as the record industry's newest formats, CD+Graphics [CD+G] and CD+MIDI [Musical Instrument Digital Interface] for synthesizers. To add to its functionality, a host of accessories, including a trackball controller and mouse, are being introduced.

Changing the form factor was only the first of a number of critical decisions Commodore made in creating CDTV. Equally important, Commodore replaced the traditional keyboard with a handheld remote control. The CDTV remote combines the functionality of a conventional remote with the playability of a video game device. While the CDTV remote is obviously limited for word processing or spreadsheet applications, its appeal is in its inherent familiarity to other living room interface devices and overall ease of use. Needless to say, when you shift the user interface from an arm's length distance of a computer monitor to a six-to-ten-foot distance of a home TV set and replace the keyboard with a remote control, screen images—particularly text—must also change. Successfully redesigning on-screen graphics is one of the challenges facing CD-ROM title developers seeking to reach the consumer market.

While CDTV is being targeted to the consumer market, it is equally appropriate for the education, library and vertical commercial markets. With parallel and serial ports for printers and modems, with ports for a wired keyboard and floppy/hard disk drive, the base CDTV player can be easily

reconfigured into a full personal computer. It also comes with a RAM/ROM front port for a personal memory card with up to 256K storage to further enhance player functionality. Rear slots allow for easy integration of hard disk, SCSI, LAN or other specialized videocard. The use of a genlock easily integrates CDTV with a laserdisc player to create a third-level, interactive, full-motion solution.

CDTV is a high-quality, affordable, all-in-one computer/CD-ROM player. Commodore is working with a host of training, catalog, point-of-sale/information, other business groups and value added resellers to develop appropriate multimedia solutions to their specialized needs.

It is in the consumer or mass market that Commodore expects to see CDTV's greatest impact. Key to its success is providing a diverse selection of titles or applications. The first CDTV catalog listed 92 titles in five categories: Arts & Leisure, Education, Entertainment, Music and Reference; Periodicals and Productivity are forthcoming. The wide assortment of titles is designed for the whole family, both children and adults. However, Commodore has made a commitment to ensure a strong representation of education, learning and reference titles. Approximately half of all current and planned titles are learning related, the other half are games, simulations and entertainment oriented. This balance is appropriate for a TV-based experience as well as the challenging decade that lies ahead.

Today, there are over 400 CDTV licensees worldwide. They are drawn from a diverse assortment of backgrounds, including many of the world's leading media conglomerates (e.g., Disney Software, Grolier, Lucasfilm), specialized multimedia publishers (e.g., Applied Optical, Discis, ICOM Simulations and Xiphias), leading games developers (e.g., Accolade, MirrorSoft and Spectrum Holobyte) and even garage-shop startups that are drawn to CDTV because of the entrepreneurial opportunities this exciting new medium makes possible. Publishing for CDTV is relatively easy and remarkably affordable, be it for an encyclopedia, a Jack Nicklaus golf game or even a way-out cyberpunk comic book.

In conclusion, CDTV is a bold initiative to bring CD-ROM to the mass consumer market. First, CDTV represents the historic conversion of consumer electronics and computing. These two essentially parallel industries are now coming together to create not only a new type of home appliance (i.e., one with a CPU as the active component of a consumer product), but also a new generation of CD-ROM applications or titles that will provide consumers with a richer, more rewarding participatory experience. Second, CDTV addresses consumers' deep-seated resistance to home computing by establishing a new category that goes beyond the limitations of both consumer electronics (i.e., single-function products) and computing (e.g., word processing or spreadsheets).

Third, the CDTV experience—like TV viewing, VCR or laserdisc usage and videogame playing—is based on an enhanced interactive relationship between the user and the home TV set. CDTV expands the current interactivity experience of TV usage: while TV viewing is (at best) reflective, VCR or laserdisc usage offers single-function involvement and videogame playing maximizes eye-and-hand coordination, the CDTV experience is participatory. Combining the multi-functionality and control of computing with the enormous storage capacity of CD-ROM, CDTV adds a new dimension—offers an enhanced experience—not available from any other home entertainment and information product.

Finally, CDTV is a product for today and tomorrow. Its diverse titles have appeal to the whole family as the next-generation multimedia CD format. But with the ability to expand to a full computer or add a modem, printer or genlock, it gives consumers flexibility and multifunctional capabilities which they can take advantage of depending on their own needs or desires. This will become increasingly important during the 1990s as we see CD-ROM being integrated into innovative consumer media

categories. Two such areas could be multimedia CD-based catalogs integrated into on-line or other telephone-network services and the creation of multimedia homework integrating text, audio and graphics/videodata drawn from a CD disc and other sources on videotape. This is the future. In the mean time, CDTV is available commercially in the U.S., Canada and throughout Europe.

User Interface Guidelines

I. INTRODUCTION

II. USER INTERFACE GUIDELINES

- 2.1 The Screen
- 2.2 Remote Control and the User
- 2.3 User Interaction
- 2.4 Selection
- 2.5 The Remote: Specification for Use with an Application
- 2.6 Other Guidelines
- 2.7 Accessories

III. USER TRAINING OF TITLE USE

- 3.1 Tutorial
- 3.2 Help Function
- 3.3 Error Messages
- 3.4 Title Documentation for the User

IV. INTERNATIONAL CONSIDERATIONS

- 4.1 Preparing Titles for Other Languages
 - 4.1.1 Allowing for PAL
 - 4.1.2 Language and Cultural Distinctions
- 4.2 Foreign Language Conversion
 - 4.2.1 Segmentation of Audio Tracks to Allow Translation
 - 4.2.2 Appropriateness of Symbols (Icons)

APPENDICES

- A. CDTV Glossary
- B. CDTV Is Not A Computer
- C. IR Remote Control Description

I. Introduction to The User Interface Guidelines

The purpose of these guidelines is to promote a uniform look and feel to CDTV applications. What does "uniform look and feel" imply? As presented here, it means that many of the Amiga intuition capabilities are disallowed as being difficult to see in a living room environment or too complex or confusing to a new home user. It means that low level issues such as a small number of fonts and the colors to be used are strongly suggested. It means that many "high level" functions, such as control panels, list requesters, and scrolling functions are to be supplied by a toolkit (or descriptions provided prior to the toolkit being done.)

The effect of these standards will be to increase the uniformity of look and feel to the user across many different applications. Note that this should have no effect on the developer's freedom of information presentation or content (i.e., an animation or graphic, which is what the user selected, will be under complete control of the developer). Also note that by supplying the toolkit pieces, the standard fonts, and other elements of the UI standard, the level of effort for developers should be greatly reduced.

The user interface for the CDTV device should reflect the intended user population and environment, that is, it should be aimed at use on televisions in a home recreational setting by unsophisticated and often brand new computer users. We are dealing with a TV user, not a computer user (i.e., entertainment not work), and should be motivated to build titles accordingly.

These user interface guidelines are presented to assist you in developing an application that is easy for the average CDTV owner to use. Remember that the CDTV player is not sold as a computer and that every time you deviate from the guidelines, you place another obstacle between your product and the user. Our competition is not other CDTV applications, it is television. With too many obstacles, the user will simply change the channel.

These User Interface Guidelines are not laws set in stone.

This document should be considered a work in progress. We invite your questions and comments. If you would like to become more involved in the continuous process of CDTV UI standards evolution, contact Alan Campagna or Guy Wright, c/o CDTV, Commodore, West Chester, PA.

Use of This Document

This document, the CDTV User Interface Guidelines, is written for CDTV developers to ensure that CDTV titles follow a consistent pattern for user interaction. At some point in the future, Commodore is likely to limit some developer benefits to titles that adhere to the standards. These might be inclusion in Commodore's booth at major exhibitions, representation on the *Welcome* disc, bundling opportunities, etc.

Equally as valuable as these pre-sales benefits are the post-sales value. Following the UI guidelines means minimized post-sales support. If a user learns how to operate one title that follows the guidelines, there will be hardly any learning curve for other titles. This not only means fewer support calls and better word of mouth PR, but is likely to result in better product reviews and fewer dissatisfied customers and returns.

It is planned to have the UI standards available in data base form. In the near future, it is planned to accompany this document with source code, IFF files, and working programs to illustrate the guidelines and demonstrate how they may be implemented.

Sections II, III and IV of this document describes the recommended guidelines.

Overview of CDTV UI Issues

The UI standards focus on issues for consideration by developers (especially Amiga developers) before and during the title design stage. Without keeping these standards in mind, a developer might have the tendency to design the title with a personal computer in mind and subsequently, at the detail level, turn to these UI notes for help - too late!

When starting a CDTV project, remember that good CDTV titles include the following characteristics:

1. Ease of use.
2. Consistency of operation. Uniform look and operation across all similar applications.
3. Minimum documentation, or intuitive operation in any situation.
4. Take advantage of the CDTV capabilities (using up to 650 Mbytes of data, including sound, music, and digitized speech, etc.)

II. User Interface Guidelines

This section presents the guidelines in the following order:

- 2.1 The Screen—The issues related to viewing distance, TV screens, colors, and fonts.
- 2.2 Remote control and the user—Use of the remote as a positioning device; limitations and suggested uses.
- 2.3 User Interaction—General and specific situations of user interaction with applications.
- 2.4 Selection—User interactions specific to selection situations.
- 2.5 The Remote: Specification for Use with an Application—Specific button usage.
- 2.6 Other Guidelines—Operation of the applications in booting up, exiting and other common situations.
- 2.7 Accessories—Support of CDTV accessories.

2.1 The Screen

Televisions are not the same as computer RGB monitors. Television is an interlaced, overscanned medium. What looks good on a monitor, may look terrible on a TV set in the home. View your screens on a TV set (both PAL and NTSC) before you commit them to CDTV disc. No matter how good your application is, if it doesn't look good on the home TV set the users will be disappointed.

Viewing Distance

Keep in mind that users will have to work with the application from a 6 to 8 foot viewing distance.

Simplicity of Screens

Screens must be simple. Colors must look good on TV. Fonts must be large. Symbols must be easily recognized, large in size, distinctive in color and shape, and meaningful to most viewers.

Fonts

All fonts should be anti-aliased, outlined or backgrounded for good visibility.

Text should be limited to a maximum of 25–30 characters per line, ten lines of text per screen.

Fonts should be sans serif, bold (but not so bold that holes in letters close up). TV screens tend to blur images slightly, filling in small holes. Avoid script, open faced, or condensed type. Some fonts that work well on TV are Optima, Theme, Futura, Metro, News-Gothic, Spartan, and Tempo.

The font should have a 1/8 to 1/5 width to height ratio.

A good rule of thumb is to leave about 1/2 letter height space between lines of text.

The best text colors are high luminance, low color saturation. Avoid fully saturated colors. Text also works better if outlined or with drop shadows or both.

The best way to test text is to try it on a TV set on the background you plan to use. If the background is busy or bright, you might try boxing the text in a neutral color.

Currently, two new fonts are available to CDTV developers from Commodore without charge or restriction.

Colors

Colors should be subdued rather than bright to minimize bleeding. On a color saturation scale of 0 to 15, colors should be no more than 12.

TVs have about a 1:20 lighting ratio (the brightest thing on a screen can only be 20 times brighter than the darkest thing). This can be further broken down into a 10-step grey scale. In order to have an object appear discernibly brighter than another, it must be at least one step away on a 10-step grey scale. In order to make something appear much bolder (such as text), it should be at least two steps brighter or darker.

Background colors should be an off-white or grey. Avoid using pale colors (pale pink, green, etc) as these can give surprising results on NTSC or PAL. For example, an attractive pale, salmon pink on a PAL TV set becomes ugly virulent orange on an NTSC set. Note also that saturation levels vary from PAL to NTSC. Colors that appear "normal" in NTSC may wash out in PAL. Refer to Developer Notes section on PAL/NTSC considerations.

You should test your artwork on both PAL and NTSC TV sets, or monitors with composite input. Contact your local Commodore office for availability.

As a rule of thumb, it is preferable to have text in a dark color on top of a light background.

Color Dependence

Options should not be presented as distinct only in color. For example, do not say "select the green box" (eight percent of the male population is colorblind to at least one combination).

Flicker

Beware of stark contrasts (all-black text on an all-white background for example). These conditions may exacerbate the flicker on an interlaced screen.

In interlaced modes, flicker is more evident. Flickering is most notable over sharp contrasts. Horizontal lines flicker more than vertical or diagonal. PAL flickers more than NTSC. Make all horizontal lines at least 2 pixels wide.

To reduce flicker, you may use “shadowed” fonts. This trick, often used on TV subtitling, is very effective.

Product Testing for TV Viewing

Perform final testing of products on a TV, not an RGB monitor. View your product from a normal TV viewing distance. Test on both PAL and NTSC.

Number of Items on Screen

There should be no more than 9 symbols on screen at one time. There may be more items at one time if they are all of the same type (e.g., numbers in a list), and easily recognizable by the user.

Borders

Screen developers should work within a “safe” area. The practical “safe text” area on a screen should have a 1/10 total screen size border on all sides. For example, a 320 x 400 interlaced display should have 32 pixel-wide borders on each side and 40 line borders on the top and bottom of the screen.

Note that screen borders may need to be changed when going from PAL to NTSC. PAL CDTV players display 256 lines (512 in interlaced mode); NTSC players display only 200 lines (400 interlaced). Use the *Bookit* utility in the startup-sequence to center the screen according to the user's Preferences settings.

Desktop Model

The home user is not in a productivity environment, and the TV screen is too small for scrutiny from 8 to 10 feet (2 to 3 meters) away. For these reasons, the following rules apply:

- No windows, no window borders—the whole screen is a window.
- No window gadgets; no windows to close or resize.
- No pull down menus. Use symbol menu trees. See Section 2.3.

Text

Wherever possible, *avoid text* for menu choices in favor of symbols.

In text presentation, design the screen for large fonts. If the screen is static in nature (such as credits), use multiple screens or scrolling text if text does not fit well, rather than reducing point size.

If the application must present scrolling text to be read, make the scroll pleasant so that the large font size can be easily accommodated. If possible, propose various font sizes, and let the user choose his font of preference. This lets users choose a large font for working on TV sets, and a smaller one (displaying more text per screen) for working on a monitor.

Note that text by its very nature is language dependent, making the distribution of the application country dependent. If text is not necessary to the application, avoid it or localize it as much as possible.

Screen Saving

Since TV screen phosphor can be damaged by constant, unchanging pictures, Commodore strongly recommends that the developers provide for a "time-out" on any screen. If user input has not been received for some period of time the screen will blank or go to a "screen saver" mode (which can revert to the prior screen on any button event).

Use the BookIt utility in the startup-sequence to invoke the system time-out screen.

Avoid "Dead Air"

Avoid black screens. One of the prime rules of TV broadcasting is to avoid "dead air", when nothing is being transmitted. This holds true for CDTV.

Avoid black, empty screens. When you must load a new image, either leave the previous image on screen, or add a message or symbol indicating action.

In order to avoid long bootup sequences, use the keeper utility in the startup-sequence to display something (a logo, perhaps) while loading an application.

2.2 Remote Control and the User

Primacy of the Remote

The standard interface shipped with the CDTV player is the infrared (IR) remote controller. All the features of an application should be accessible through the remote controller.



Applications must be designed to use the remote. Do not assume any other device. When the keyboard or joystick is required for the application, the remote should still function properly.

Most users will have only the remote. Design the application so that the remote can be used for everything.

Positioning—No Screen Pointer

As a positioning device, any remote unit will be awkward to use for fine adjustments. Moreover, like a mouse, it is not completely intuitive. For that reason, the guidelines are to have:

- No pointing of a cursor.
- No gadgets for resizing, dragging etc.
- No double-clicking.
- No dragging.
- No pull down menus.

This means simplifying screens, limiting the number of options on each screen, and supplying defaults wherever possible.

Key Functions

A set of definitions of key functions has been established. These are presented in section 2.5.

Mouse/Joystick Selection

If the operation of the application depends on this being in one state or the other, make sure the user is prompted to set it...try to determine that it is correct.

In general, do not depend on the position being correct, and adjust the software to operate well *in either position*.

2.3 User Interaction

The visual rules in 2.1 and the use of the remote in 2.2 are augmented here to describe the standard operation in dynamic or interactive situations.

The keys to the rules in this section are consistency and the use of visual and audio cues to better support user interaction.

Feedback

Both visual and audio feedback are *necessary*. When the user presses any button which can be interpreted by the application as meaning a change of state or user action request, *let the user know that the button was read*.

When the user has an item highlighted and presses one of the select keys on the IR remote, have a graphic or audio indication that the program got the message. A "busy", "working", audio beep, screen flash, fade to black, or other visual/auditory signal will prevent users from pressing the select key over and over or thinking that the player is broken. The button should highlight along with a simultaneous audio "beep".

When the options are complex or detailed, provide help screens or audio help.

Negative Feedback.

When the user presses buttons which are irrelevant to the process at hand, an audio cue that this is "wrong", such as an unpleasant sound, would greatly help to channel the user in the right direction. If it is necessary to have buttons that are inoperable or irrelevant, those buttons should be ghosted and not selectable.

Disc Wait Feedback.

Whenever the application must access the CD, display a visual cue (busy icon) so the user knows something is happening. Inform the user when disc I/O or processing is going on. Something like animated turning gears, "Zzzzz", or a "Working..." message might be suitable. Try for animation wherever possible. Commodore will provide some default recommendations, but title designers may wish to use something more specific to their context.

Animating the book selected (such as making the book move out from the shelf) is a good way to let the user know that the button select was seen and to keep the user's attention while the next screen is fetched and made ready for presentation.

Limitation of Activity

Maintain a single window of activity for selection operations, or ones which require the user to perform an operation. The multi-window environment is too confusing for CDTV users. (Note that this does *not* mean that you should avoid multiple windows for presentation; these can be very interesting and useful.)

What should be avoided, is the necessity for the user to choose the process to which to send an input.

For example, a typical Amiga Workbench situation may have several active windows on the screen at the same time; the user indicates the window to which he/she wishes to input data by clicking on some portion of that window. This is what should be avoided in CDTV screens. Multiple requesters, such as those that appear in the CDTV ROM Preferences screen, are acceptable, since the user can clearly position to the requester desired with the arrow keys, so there is no confusion of where the next key strokes are going.

In some applications, such as games or those in which user input is treated as asynchronous with the application flow, care must be taken to keep the user from being confused as to what is expected and what effect pressing the button is having.

Consistency of Operation

Ensure that symbols (icons) have a standard meaning for button operation in similar situations.

Section 2.5 presents standard uses of the buttons.

Menu Design

Selection situations should always look the same; the look may change, but feel and function are the same.

- Use a symbol cycle, described below.
- Use ghosting to indicate an option is not currently selectable.
- Always provide a uniform exit symbol to return to upper most level of a menu tree (main menu).
- Support a standard way for user to navigate menu trees.

Refer to Section 2.4 for more details on selection operation.

Symbol Cycle

The selection by the user of the next action to perform should be done by having the user cycle through a list of selectable options represented as symbols (subject to the limit of 9 symbols per screen). Always indicate the currently selected symbol. Outline, frame, flash, reverse the background color or animate the items as they are highlighted. The user must be able to see which items are highlighted from across the room.

Provide symbols on screen to perform most operations. Do not replicate remote control buttons on screen as symbols.

The selected symbol is then chosen with the OK (enter) or A button.

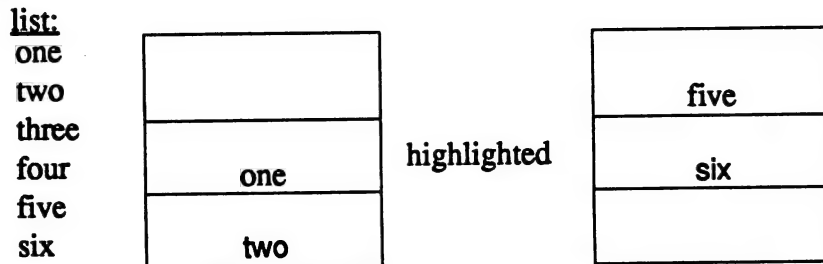
Provide a simple cycle algorithm so that it is obvious how to get to any symbol quickly. Always provide a default symbol. Attempt to choose the most appropriate default. The default should be the symbol the user will most likely choose next.

Make the transition cycle smooth; control the "bounce" so that the user does not have to hold the button too long nor overshoot desired symbols.

Selecting from a list

Selecting from many choices (words, numbers, or icons) is accomplished by presenting a list which scrolls vertically. The list scrolls behind a highlight. Using the arrow keys, the user moves the list, making the highlight appear to move. Specifically, the *left* and *up* arrows move the list *down* (the highlight appears to move up the list), and the *right* and *down* arrows move the list *up* (making the highlight appear to move down the list). To reiterate, we are talking about moving the list up and down. The center item always remains highlighted. The user scrolls the list behind the highlight.

Except in cases of extreme space constraints, a minimum of three list items should appear, displaying a blank at the top when the first item in the list is highlighted and at the bottom when the last item is highlighted. Here is an illustration:



This gives the user a visual clue when there are more items to choose from than can be displayed at one time.

In long lists, accelerate the list scrolling by holding down the direction arrow for more than a few seconds.

To indicate a choice, the ENTER or A (left select) button is pressed when the desired item is highlighted. In a single-choice situation, pressing this button takes the user out of the list window and onto the next step.

The B button exits the list without making any selection, i.e., acts as an abort.

When more than one choice can be made from a list, the A button does not exit the list but changes the highlight. The changed highlight remains on the chosen item if the list scrolling is continued. Placing the chosen item in the center of the window (i.e., the regular highlight area) and pressing A again removes the special highlight, i.e., de-selects.

To indicate that selection has been completed and to exit the list, one or more “done” or “end” words or icons should be included in the list. For example, to select two items (“three” and “four”) from the list:

list:
one
two
three
four
five
six
done

two
three
four

press ENTER or A

highlighted

three
four
five

press ENTER or A

special
highlight

six
done

press ENTER or A

highlighted

If appropriate, the PLAY button could also be used at this point to exit the list and begin an operation based on the section.

If the order of selection is important or an item can be selected more than once (or if the list is long enough so that it is difficult to keep track of choices), a separate “item order list” is built as each item is selected. (ENTER, A, PLAY, and B buttons act as above except that choosing an item a second time does not de-select it. Instead, the item is selected a second time. Using the previous example and selecting “three-four-three”:

two	highlighted
three	
four	

press ENTER or A and item order list shows: three

three	highlighted
four	
five	

press ENTER or A and item order list shows: three
four

two	highlighted
three	
four	

press ENTER or A and item order list shows: three
four
three

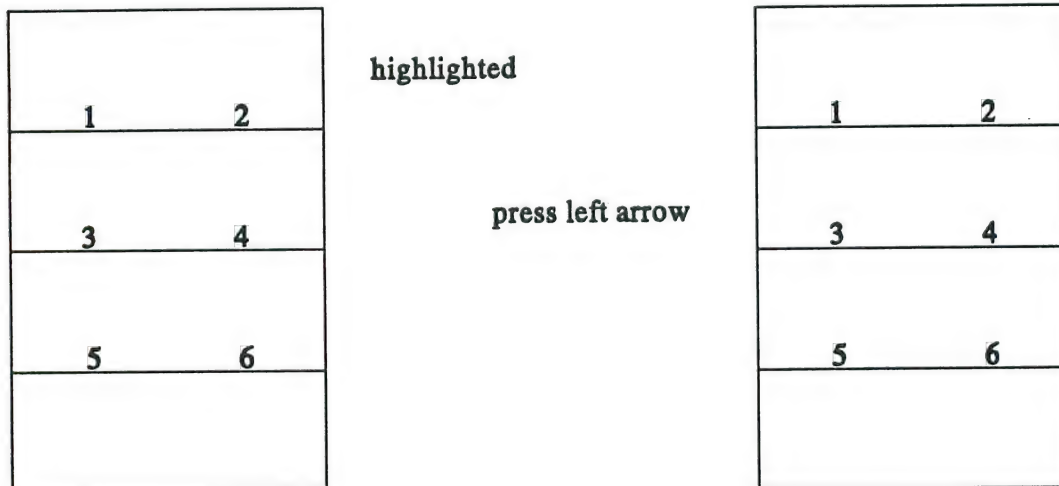
The list could include separate edit items such as “clear list”, “delete last entry”, etc. Alternatively, these edit options could be outside the list. An “edit” item in the list would move the user to them.

Matrix, Calendar, or Grid Lists

When the choice is icons or numbers (not words), this style is a viable alternative. (The track selection of the audio control panel is an example of this type of list.)

When there are more entries than can be displayed at one time, the entire grid scrolls up and down (that is, row by row) when the highlight is moved above or below the items displayed. Except when it is obvious (as in the track selector which begins with "1"), a blank row should be used top and bottom as with the vertical scrolling list to indicate when there are no more choices. At least three rows should be displayed.

With grid lists, the highlight is different from the vertical scrolling list. In this case it can be moved around the grid in any direction with the arrow buttons. The highlight moves in the direction of the arrow pressed.



Pressing ENTER or A button indicates a choice and exits a single-choice grid. When appropriate, it also begins the selected function. A multiple-selection grid requires a "done" option and acts as described for vertical scrolling lists, either changing the highlight or adding an item to an order list. B aborts the process. PLAY begins execution of the related activity (if appropriate).

Symbol Design

Make each option as clear as possible, from the design of the symbol.

The best symbol is clear and intuitive to the user—it does not have to be explained.

Symbols are better than text.

The average CDTV owner does not have the usual computer literate understanding of symbols and visual cues which are standard in the industry. Test your symbols on non-computer people first.

Use interesting symbols, specific to a situation, wherever possible. For instance, the "bookshelf" selector is a good one because users can relate easily to "choosing a book"; this gets them out of thinking of the machine as a computer and moves them into a familiar environment.

Help Function

Provide help at all levels of the application.

Build help into every menu selection.

Use the Escape “?” button to bring up a help screen.

The help screen can be used to allow the user to enter a tutorial on the product. This prevents the user from having to navigate back to the top (or wherever the tutorial entry point symbol was) to find out what to do.

The help screen can be used to provide an entrance to extended functionality, or special navigation of the tree, or setting of Preferences and modes. This allows the power user to get to things quickly without bothering the novice user with extraneous decisions.

Escape Function

Use the Back button “B” to allow the user to get out of a section of the menu, back up a level in a tree structure, or to stop a presentation and return to the menu which activated it. This is a very important user friendly feature which most users will come to depend upon. It allows them to audition sections of the application without running the risk of having to sit through presentations which they don't want to see.

2.4 Selection

Selection situations are the most common application requirement. Users must choose areas of interest, actions to be performed, or set Preferences or options. Each situation is somewhat different, yet the similarity of needs leads to the following general set of guidelines. Standardization of the selection rules will help all CDTV developers by giving CDTV users the confidence that they can pick up a new product and use it successfully without hours of training.

Selection Modes

Use a single click on the left select button (marked A on the remote control) to activate an option or select an item. (Technically this is the equivalent of pressing the left mouse button once—no double clicking.)

The selection methods which can be used, in order of preference;

A menu of fixed items

This is by far the most useful form of selection. It may take the form of a tree of fixed menus, or a linear chain of connected menus.

Lists of similar items

Here the items represent similar objects. These objects may be (for example) the countries of the world or the food topics covered in a group of recipes. See below for a presentation of choice of lists versus menus.

Browsing

Although this may be accomplished via menus or lists, browsing can also be supported through visual analogies, such as geographic movement, or simulation of motion through space.

Choice of Selection Mode

To choose between the use of the selection modes (menu networks, menu chains, or lists), the following guidelines are suggested. Each mode may be appropriate for a section of the application. The modes may be mixed, but the general order of use follows.

Menu Networks

Use menu networks for top level choices, and for all action choices. Lists are seldom used for action choices. The Welcome Disc is a menu network. Most application selections can be handled with menu networks.

Menu Chains

Where a menu tree goes beyond 2 or 3 levels, it gets too complicated to remember where you are. To keep the number of levels small, break large numbers of choices into a menu chain, connecting each screen to the prior and the last with arrow symbols. At any level in a menu network, where the number of choices is larger than 9 items on a screen, make it a chain at that point. The linear form is better for equal level choices such as language Preferences. Tree form, which is most general, is preferable where a natural hierarchy exists.

Lists

Lists are used mostly for selection of objects, not action. Each object bears a similar relation to the others in the list (e.g., they are all planets or bodies in the solar system). Mixing several types in a list is confusing and usually indicates that several lists should be used. The time, track, and language selections in the system audio control and preferences are examples of list selection.

The order of preference is therefore: menu network, menu chain, list.

Text Entry

Where absolutely necessary, text entry can be accomplished with the remote by:

- a) Selection from a "hot word" list, or making words in text already presented hot.
- b) Presenting the user with a keyboard on the screen which treats the letters as selectable symbols. (The arcade name selection technique presents the letters as a linear, alphabetized list).

The discussion concerning the language dependence of any text-based application applies more so to text entry.

Guidelines for Menu Networks and Chains

Limit the number of symbols to 9 on screen at one time.

Provide feedback (see Section 2.3) for indicating selected symbol, and button presses.

Provide help on any screen: this may be a specific symbol, (standard help symbol is "?"); or use the Escape button.

Provide network control symbols: up to move to the last level, right to the next in the chain, left to move to the last in the chain. Audio forward-back button may also be used for flow.

This leaves about 4-6 "business" symbols on most screens, which is a determining factor in limitation of menu depth.

Selection of menu symbol: To select an item, activate an option or begin a task or function, the user will press the left select button marked "A", or the Enter button. The button will be pressed once.

Navigation aids: in addition to the network control symbols, provide some form of "map" for the user, letting them know where they are in the tree. This can be done by appropriate screen design, or titles or symbols which let them know where they are. The Help function can contain a brief description of where the user is.

Guidelines for Lists

Lists may take many forms. A list should have objects which are related to one another.

Scrolling is a natural function for lists. The nature of the scroll function is critical to the pleasant use of the list.

Refer to the ROM interface for examples of different types of lists:

Numeric List, by digit

The time setting of the ROM Preferences shows how a number list can be scrolled by using the up/down arrows for list scroll, with the Enter button for selection of the item.

Item List

The Track Select shows how items can be presented in a grid form, allowing the directional arrows to be used in both horizontal and vertical movement. In the case of the grid being larger than the screen, a directional arrow appears when the user positions the selection to the top or bottom rows; if there are more selections above or below, the arrow indicates that an arrow movement "off" the grid will cause the grid to scroll. Refer to the ROM Audio panel for examples. Note that the Audio panel operation also illustrates the requirement for selection of multiple items from a list. This is covered below.

List as a menu chain

Note that the Preferences operation of the ROM handles the selection of language as a fixed menu of items, chained together. It is worth noting that when the items of a list are fixed, as they are in this case, the use of a linear menu chain is much easier on the user and preferred to a general list format. Grid lists can be used where the selected items can be easily seen and recognized by the user. For example, numbers and single letters are recognized rather than read. Grids are not appropriate for text, or symbols which are not easily recognized.

2.5 The Remote: Specification for Use With an Application

The use of the remote has been mentioned in the prior sections. Here we describe the use of each button, and additional features of the application which should be triggered directly from the remote. Two general rules should always be followed:

- Observe standard uses of the keys.
- Do not duplicate key functions with symbols on the screen.

"A" Button

This is the primary "Enter" button. It activates selections in menus and lists. Since it is usually under the user's right thumb, it is easier to reach on the remote than the Enter button. It will be used almost exclusively for most applications.

Enter (OK) Button

In most situations it functions the same as "A", but may have alternate meanings in multiple selection situations. Refer to the Audio Panel track selection operation for an example; the "A" button means "choose this item and add it to my list of selections", whereas the Enter button means "I'm done with the list".

The distinction between the use of "A" and Enter is that 1) where two forms are needed, as in multiple selection, "A" is the intermediate "OK", and Enter is the final "accept all" indicator; 2) in situations where data will be saved (e.g., bookmarks or game scores saved to NVR) the Enter is required to force the user to certify that this is the action desired, since the "A" button is too easy to hit accidentally.

"B" Button

This button functions as "Exit this level", meaning "Go up to last menu."

"B" is used to break out of a section or level. It should also serve as an "abort operation" button, to terminate an animation or presentation and return to the menu which caused the action to occur.

Do not use the "B" button to access pull-down menus (as in Amiga software). Avoid pull-down menus entirely.

Escape Button

The Escape button should be treated as a "?" (question mark.) Its function in all cases should be to bring up some kind of help screen.

The help screen might be an access portal to many other features of the application; it could give the user the option of going through a demo or tutorial section, refer the user to her manual, bring up contextual help screens, or have special options such as "return to top", "return to previous screen", "jump to another section", "jump to another mode", etc. It could allow the user to change operational modes such as power user functions. This can be the primary means by which the user learns about the features of the application.

The distinction between the "B" button and the Escape is that the "B" is more simply a menu navigation and presentation control button, whereas the Escape is to provide help and let the user know more about what to do.

A/B Buttons In Game Applications (ONLY):

They are the equivalent of the left and right mouse buttons in mouse mode. In joystick mode, only the left selector button will function as a fire button.

Note to game developers: the remapping of the remote buttons for special needs of games is certainly possible, but the standards for such uses are beyond the scope of these guidelines.

Arrow Buttons

On the left side of the remote is an eight-way direction key, which is used to move pointers on the screen, make selections, etc.

These arrow buttons are always used to move around. Move from symbol to symbol, move through a list, move to another selection, etc.

Joystick/Mouse Toggle Button

When in the joystick mode, the direction and fire button signals sent to the CDTV player will be in standard joy, four button, eight-way form. When the remote is in the mouse mode, intuition mouse movement calls will return values in four pixel increments rather than normal Amiga one-pixel increments. This was done, primarily to enhance the responsiveness and speed when moving a pointer. Developers should try and write their applications to trap for both modes if possible and notify the user to press the joy/mouse button to change modes if necessary. A simple "press left selection button to start" message to the user will let you determine what mode the remote is in. This button, currently a toggle button, will become a switch in the next revision of the remote control. The user will be able to identify the mode he is in by simply looking at the remote.

When in joystick mode, the only active buttons on the remote are the "A" button and the four directional arrows. All other buttons are disabled. Should your applications need any other buttons, tell the user to switch to the mouse mode.

Numeric Buttons

Numbers are entered from the remote control using the number keys. ENTER indicates that a number is complete. For example, on the audio panel, the user is able to enter individual track numbers in this manner. To indicate that they have completed a track number, they press the ENTER button, e.g., 6—ENTER (for entering track 6 in the list) or 2—9—ENTER (for entering track 29 in the list). This avoids the necessity of requiring leading zeros on single digit tracks.

Audio Control Buttons

Next to the volume controls are buttons for controlling all the audio functions of the CDTV player (rewind, fast-forward, play/pause, and stop).

In addition to audio control, the audio buttons on the remote control and the front panel of the player should be used as "navigational aids" for the user.

Play/Pause

A press of the play/pause button pauses the application. Another press re-starts the application from that point. Play/pause also acts as a mute button, halting the audio as well.

Fast Forward

The fast forward allows the user to skip forward through screens or a section.

Reverse

The reverse button allows the user to back up through screens or a section.

Stop

The stop button may be used to stop an application. However, beware of inadvertent keypresses on this button. If stop is used, the Play button should re-start the application from the beginning.

Obviously, there are some applications where use of the audio buttons for these purposes doesn't make sense, but in many cases the extra controls given to the user can simplify things quite a bit, since their use in the audio world is well understood.

GenLock Button

For use with the optional genlocking accessory. This button cycles the CDTV player through the three genlock modes: source only, mixed source and computer, and computer only.

TV/CDTV Button

For switching between CDTV and normal TV viewing when the CDTV player is connected to a TV via an RF modulator (this toggles the RF modulator and light on the player front panel).

Volume Buttons

These are next to the A/B selector keys. There are two buttons for adjusting the headphone level volume up and down. The volume level is indicated by the light on the front of the player.

Power Button

Turns the player on and off.

NOTE: if the player is turned off with the remote, it must be turned on with the remote, even if the panel on/off switch is used.

NOTE: all applications may not be able to use these buttons, but when practical they simplify operation since they are well understood by users.

2.6 Other Guidelines

These issues relate to the operation of titles other than the interactive ones covered previously.

Documentation

There should be minimum printed documentation. A well designed application will result in intuitive operation in any situation.

Timeouts

Build "timeout" features into applications. When there is no input from the user after a few minutes, the program should revert to a self-running mode. This is particularly important at the beginning of a session. Startup screens should offer the choice of beginning right away or proceeding through a tutorial. After a minute or two without user response, jump right into the tutorial.

Use "arcade style" demo of the product.

Refer to "Screen Saving" in Section 2.1.

Exiting

This is a minor issue in most cases because the machine will re-boot when a new disc is inserted (unless the program has specifically requested an override of the re-boot function for multi-disc applications). However, all applications should exit cleanly and if any settings or Preferences have been altered, they must be reset before exiting.

An application should never return to the CLI window. If the application must exit, it should allow the system to reboot.

Note that a program designer should trap to prevent the user escaping through the top of the menu tree by excessive "B" button presses. A simple way to do this is to make the default on the top menu something other than the exit symbol, and to explicitly check that "B" on the top menu is not accepted as a legal command.

Program Crashes

Certainly no developer will plan to ship a product with known bugs. However, some bugs do (rarely, we hope) go undetected until users do really unexpected things with the software.

Plan to have a mechanism to get information on what the user did to cause the problem.

In the case of system crashes, if applications use the standard 1.3 operating system provided in the ROMs, the guru meditation has been removed. In case of a fatal error, the system resets.

Autobooting

All applications must be auto-booting.

This can be simply accomplished by providing the usual startup.sequence file in the system S: directory. Do not hardcode your startup sequence to a specific device number.

The BookIt utility, available on the CDTV Tools diskette, should be included in your startup-sequence. It reads the user preference settings from the NVR, and centers the screen accordingly. BookIt can also be used to implement a screen saver.

Nationality and User Preferences Selection

Ideally, all applications will be either language independent or provide versions of the program in all languages. Developers should be aware that this machine will be sold in many countries and when the application does not currently support the language selected in Preferences, the user must be presented with alternatives.

While many people will adjust the Preferences (particularly nationality settings), most people will never touch them (if it ain't broke don't fix it). However, certain adjustments will have to be made by the user upon power up or after a loss of power.

One of the very first things that an application should check for is a match on language settings in Preferences. If the application does not provide a language specific version matching the particular language selected in Preferences, the application must provide the user with an alternate choice or set of choices.

2.7 Accessories

- **Printing**— If an application requires printer control, developers should contact Commodore for printer drivers and Preferences code. There is a location in the NVR to store a printer selection, however, the printer drivers must be included in the title. When an invalid value is found in the printer save area in the NVR, the title should bring up a screen where the user can indicate the type of printer in use. This means that the first time a user selects “print”, the title currently in use will put up a screen to set the printer Preferences. This can be done through the *prtprefs.library*.

III. Help Within Titles for Users

3.1 Tutorial

All applications should provide any necessary instructions on the disc itself. This can be done at the start if desired, but a means to escape the tutorial must be included. A link to the Help function should be provided at all times in the title.

3.2 Help Function

Applications should provide Help screens or audio help for the user. The Escape button has been designated as the "Help" button (see Section 2.5).

When the application does not require Help screens, bring the instruction screens back up or deactivate the Help button function.

3.3 Error Messages

There will be a standardized set of error messages to communicate to the users when they encounter problems such as: Personal Memory Card full, file not found, Memory Card not present, printer trouble, etc.

This error functionality is to be provided as part of the developer support code to be distributed to registered CDTV developers.

3.4 Title Documentation For The User

Documentation should be an integral part of the title itself. The best titles do not require hard copy documentation.

IV. International Considerations

It is important to keep in mind that much of Commodore's business is outside of the United States, particularly in western Europe. It is expected that a similar sales distribution of the CDTV player will occur.

Consequently, it is important to consider international aspects such as language, local customs and current success stories (i.e., what is "in" in the countries you are targeting?). Titles with worldwide appeal, which can be adapted easily to foreign environments, can expect far greater success than those developed exclusively for a single market.

4.1 Preparing Titles for Other Languages

4.1.1 *Allowing for PAL*

Refer to PAL/NTSC issues in Section 2.1.

- PAL has 512 lines in interlaced modes, NTSC has 400. This requires more memory and involves planning for a different aspect ratio.
- PAL is 50 Hz, NTSC is 60 Hz. This means that high resolution flickering is more prevalent especially if the contrasts are too high.
- A different color scheme may be required; PAL colors do not look the same as NTSC. Color saturation levels are particularly affected.

4.1.2 *Language and Cultural Distinctions*

- Many cultures are historically old and therefore rich in tradition; language is frequently a source of pride and great store is placed in proper use of the language. Do not assume that applications can be mechanically translated into another culture just by looking up the words in a dictionary. In addition, pictures and symbols which are obvious to one culture often make no sense (or worse, are insulting) to other cultures.
- All spoken words in an application need to be considered for translation to foreign languages.
- All titles should be tested with a native speaker who is CURRENTLY residing in the target market.

4.2 Foreign Language Conversion

4.2.1 *Segmentation of audio tracks for translation*

Segmentation of audio into pieces which can later be replaced with foreign language versions helps greatly in facilitating the process later. This approach was taken in developing the Welcome Disc. Note that synchronization of translated tracks with graphics which have distinct time dependencies can cause some problems, since the translated narrative may take much more time than the original (e.g., English to German). In printed form, most European languages occupy 25-30 % more space than English (i.e., it takes more words to express the same thing).

4.2.2 Appropriateness of Symbols

To avoid translation of on screen text, the use of symbols is strongly recommended (refer to Section 2.1). Note the comment above as to cultural relativity; symbols may also require translation from one culture to another. The standard symbols (Section 2.3) should remain constant, but application specific symbols should be examined during the translation process.

4.2.3 Length of text

If you choose to use text for keywords in menus, remember that a word in English requiring 4 letters (such as EXIT) may require 6 or more in other languages (SORTIE in French, for example). This is another reason to consider using symbols as opposed to keywords in menus.

Appendix A: CDTV Glossary

The CDTV glossary identifies terms which should be used by developers in building CDTV titles. These terms should be used as part of application text or narrative presentations, or in any documentation which refers to the subjects as noted.

The glossary is presented alphabetically by term. Each term appears in the left column with its definition in the right column. Preferred or alternative usages of the term will be presented in bold letters—do not use the bolding in your title and documentation.

The glossary will be updated as necessary. It is available in numerous foreign languages.

"A" button	left button on the remote used to select an action; can also be referred to as select button , activate button or fire button .
ac powercord	use powercord .
accessories	CDTV player optional add-ons, not part of basic configuration (e.g., joystick, typing keyboard, etc.)
action	user does something.
activate	begin a non-interactive process such as starting a demo; (do not use to indicate power/turning on the machine itself).
animated	graphics displayed to simulate motion.
application	a CDTV application; use title .
arrow	one of the directional indicators found on the remote control and the optional keyboard; use as up , down , left , right .
audio player	use cd-audio player .
"B" button	the right select button on the remote control; used to return to, or back up , to a previous choice or level.
begin	User starts an interactive title. For audio, use play .
bookmark	an entry in non-volatile RAM (NVR); typically used by the user to save some information such as the status or high score of a game, or by a title to keep track of an event, choice, or position.
boot	use start or restart .
browse	move in a non-linear fashion through a title.
button	a raised switch found on the remote control and other accessories except for the optional typewriter or musical keyboard where the buttons are called keys; buttons on the screen are called symbols.
caddy	carrier or case into which a cd is inserted to be played in the CDTV player.

cd	use cd-audio for audio discs, or CDTV disc for titles.
cd-audio disc	compact disc with audio only.
CDTV	Commodore Dynamic Total Vision; use only as an adjective, not as a noun; The CDTV player is correct; The CDTV is not.
CDTV disc	compact disc specific to CDTV players.
CDTV player	the CDTV system.
channel	a TV channel or audio channel (right and left stereo).
choice	an option available for user selection.
choose	use select .
clear number	remove a number that the user entered.
clear screen	remove all user selections from the screen.
click	use press for keys, push for buttons, select for symbols.
compact disc	use cd-audio disc or CDTV disc .
compatible	accessories that will work with CDTV players; other players that will run CDTV discs.
composite video	an output signal combining video and audio
composite video port	socket on back of CDTV player, into which a cable is inserted to connect a composite TV or monitor or VCR.
computer monitor	alternative display device to a TV.
configuration	CDTV player and the accessories that are required for a particular title.
connect/disconnect	to plug in ... unplug; applied only to hardware.
control panel	the plate on the front of the CDTV player where the CD audio controls (Fast Forward, Play/Pause, etc.) are located.
cover	the front panel piece over the slot where the personal memory card is inserted.
cursor	use pointer .
cycle	loop through a variety of options.
data	uninterpreted elements of information.

default	pre-determined settings; the user can override these.
device	use accessory.
direction arrow	one of the left/right/up/down arrows on the remote control.
directory	avoid this term.
disc	proper spelling to be used when referring to a cd, i.e., disc is a read-only medium, whereas disk is a read-write medium.
disk	any read-write medium, such as a floppy or hard disk.
disk drive	mechanism for reading disks.
diskette	can also use floppy disk
display panel	time/track indicators on the front of the CDTV player.
done	user has finished making a list or set of selections.
drawer	avoid this term.
eject	make the caddy or personal memory card pop out.
elapsed time	the length of time passed since the start.
enter	begin a user selected process or accept user input such as a number.
enter button	May be referred to as the OK button.
escape	avoid this term.
Escape button	May be referred to as the Help button.
exit	go back a level in a menu structure or to leave a context.
external device	something attached to a CDTV player; use accessory
fast forward	move quickly through a cd-audio or through a CDTV title.
file	organized collection of information.
fire button	the A button on the remote controller when in joystick mode or a button on a joystick; used to fire in an arcade style game; only use this term when in joystick mode

flash	turn color of an object or light on and off, or alternate colors to indicate selection.
floppy disk	a read/write storage medium used in a floppy disk drive.
floppy disk drive	CDTV accessory that reads and writes floppy disks.
format	prepare a floppy disk to have information stored on it.
genlock	a CDTV accessory which can display CDTV screens on top of video images.
graphics	information displayed pictorially.
guru	not relevant for CDTV players as all errors should be trapped.
headphones	CDTV accessory to individually listen to audio.
help	detailed directions provided to user when requested; usually accessed via the ? button.
help button	button on the remote controller used to provide help information.
help screen	the information displayed by an title as a result of pressing the help button.
highlight	display selections and options in inverse video or with a different border or other means of differentiating it; first the user highlights a choice with the arrow keys, then selects it with the "A" button.
hit	use press for keys, push for buttons.
hook-up	use connect .
hot-link	means of connecting a word or phrase to all other relevant occurrences of that word or phrase within an title. Can also use hot word .
hypercard	avoid this term.
hyperlink	use hot-link .
hypertext	use hot word .
icon	use symbol .
information	data put into a format to give it meaning.
infrared	the part of the light spectrum emitted by the remote control. Can be abbreviated as IR.
input	use action for verb, information for noun.

insert	place a cd in its caddy or the caddy in the CDTV player or a personal memory card in its slot.
instruction	explain usage to the user.
interactive	ability of user to influence or control what happens in a title.
inverse video	opposite colors such as negative and positive in the photographic sense; use highlight .
IR	abbreviation for infrared .
item	that which is selected in a menu or list; see also symbol .
jack	a type of connection most often seen in audio.
jewel case	plastic box in which cd audio or CDTV discs may be shipped.
joystick	CDTV accessory used as a remote control, usually for game playing; also a mode option for the CDTV standard remote control.
jump	go to another action, screen, symbol, etc.
key	buttons on the optional typewriter or musical keyboard.
label	unique name for a symbol .
landscape	graphic that is wider than it is tall which is displayed on its side.
left select button	A button on the remote control; can also use select button .
lesson	use instruction .
light	indicator on the CDTV player; green light means power is on, amber light means disk is being accessed.
link	to cross-reference non-consecutive images or words.
loop	to go through a sequence and then start again from the beginning.
main audio control	the play/pause, forward/reverse, stop and volume controls on the front of the CDTV player (as opposed to those on the remote controller).
main power button	the on/off button on the front of the CDTV player (as opposed to the one on the remote controller).
memory	use this term, not RAM .
menu	display of available choices.

MIDI	Musical Instrument Digital Interface; a standard means of connecting electronic musical instruments to one another or to other devices, such as a CDTV player.
MIDI in/out ports	sockets on back of the CDTV player, used to connect electronic musical instruments via MIDI.
modem	CDTV accessory enabling communication with another computer system via telephone lines.
monitor	alternative to TV for display of CDTV titles.
mouse	CDTV accessory enabling movement and selection of items on the screen.
music keyboard	accessory involved in music; a piano-style keyboard.
NTSC	North American television standard; standard NTSC resolutions are 320x200; 320x400 (interlaced); 640x200; 640x400 (interlaced); do not use this term in user documentation.
output	any signal sent out of the CDTV player to an accessory such as the signal to a printer or MIDI instrument.
panel	use control panel.
PAL	European television standard. Standard PAL resolutions are 320x256; 320x512 (interlaced); 640x256; 640x512 (interlaced); do not use this term in user documentation.
parallel printer	one type of printer accessory that can be connected to the CDTV player.
pause	interrupt a title and hold at point of interrupt so that the title can be resumed from that point.
personal memory card	a battery backed-up memory card that can be inserted into a slot on the front of the CDTV player and used to store information.
picture	what is displayed on the TV screen (e.g., clock picture).
play	begin/start cd-audio and, in some cases, a title.
player	use cd-audio player or CDTV player.
pointer	indicates position on the TV; may be an arrow or other appropriate symbol.
ports	outlets on the CDTV player for connecting accessories; most on the back of the player.
powercord	cable to the source of power.
power button	on/off button on remote control or CDTV player.

preferences	the user specified settings for clock, screen centering, printer and national language.
press	applies to keys; buttons are pushed, neither is hit.
previous screen	the display immediately preceding the current one.
printer	CDTV accessory for obtaining hardcopy.
program	use title.
push	applies to buttons; keys are pressed. Neither is hit.
RAM	use memory
randomize button	allows audio tracks to be played in a nonconsecutive order; order cannot be user selected.
reboot	use restart.
remote controller	device with buttons that sends directions to the CDTV player via infrared signal.
remote other buttons	buttons on the remote other than the A and B buttons and the four arrow buttons e.g., Help, Play/Pause.
repeat	go back to the beginning of a process just completed and do it again.
reset	reestablish initial settings or restart a title.
restart	begin again with initial settings.
retrieve	recall information from memory or disc or personal memory card.
review	browse available options such as a menu, or look over previous steps or selections.
rf converter	accessory that allows switching from CDTV to TV signals when TV is connected to the rf signal port of the player.
rf signal port	one of the video output sockets of the CDTV player.
rgb signal port	one of the video output sockets of the CDTV player.
s video signal port	one of the video output socket of the CDTV player; sometimes referred to as Super VHS.
screen	what is displayed on the TV or monitor.
screen blanker	use screen saver.

screen saver	a function that avoids displaying a static image on a TV screen for long periods of time to avoid phosphor burnout.
scroll	move through a list on the screen.
search	look for information based on a selected set of criteria.
select	choose from available options; the user highlights his choice with the arrow keys, then selects with the A button.
select button	indicates or activates a chosen option; the left or A button on the remote.
self-running	non-interactive title.
sequence	order, such as alphabetical or chronological.
serial printer	one type of printer accessory that can be connected to the CDTV player.
set (to)	define as a specific value.
skip backward/forward	move non-linearly; use jump, or next or previous.
smart card	use personal memory card.
software	use title.
start	use begin
startup screen	initial CDTV display before a disc is inserted.
stereo	refers to external sound system or audio output of CDTV player.
stop	halt current function; opposite of activate.
switch	physical on/off toggle or move to an alternative such as another screen.
symbol	graphic illustration of a choice which the user may select (e.g., track order symbol).
television	use TV.
title	CDTV application.
toggle	alternate between two positions or options.
track	section on a cd-audio disc; each track contains a song or sound.
track segment	part of a track.

trackball controller	CDTV accessory enabling movement and selection of items on the screen.
typing keyboard	CDTV accessory used for alphanumeric entries and title navigation.
tutorial	use instruction.
TV	television; acts as a display device for the CDTV player.
TV screen	portion of the TV where CDTV information is displayed.
user	interactive participant with the CDTV player.
volume	refers only to audio; increase and decrease.
wired mouse	use mouse.

Appendix B: CDTV Is Not a Computer

At the Amiga Developers conference in Atlanta in the spring of 1990, a number of CDTV seminars and forums were offered, and all of them were very well attended.

The Number One question or misconception about CDTV players that was heard from many Amiga developers centered around one point: the notion that a CDTV player is just a modified Amiga with a CD-ROM drive. Let's state at the top that a CDTV player is not just a computer with a CD-ROM drive.

Although a CDTV player has the Amiga chip set inside and much of the functionality is based on Amiga technology, it is very important for developers to keep in mind that CDTV players will not be sold as computers. The people who will be using CDTV players will probably not be computer literate, nor will they be willing to learn a lot of computer concepts. If they wanted a computer, they would buy a computer, not a CDTV device.

One of the central ideas behind the CDTV device is that it will be used in a living room environment by noncomputer people for noncomputer activities. It is not a Commodore oversight that the device won't have a keyboard or mouse shipped with it. This is done intentionally. We are trying to appeal to people who get easily confused by computer jargon. They may be techno-curious types, but for the most part they are buying an entertainment, educational, reference machine.

To put that in real terms that will be meaningful to developers, let's look at some of the user interface and environmental considerations.

Viewing Distance

The CDTV player will probably end up on top of the living room VCR. The device will be feeding video to a normal television set (*not* an RGB monitor). People will put in a CDTV disc, walk back to their easy chair or couch, pick up the IR remote controller, and begin. This means that they will be too far away to read small text on the screen. Normal Amiga fonts are far too small. Normal Amiga symbols (icons) are far too small. Anything at the edges, top, or bottom of the screen may not be visible depending on how well their sets are adjusted. Most colors will be poorly adjusted, so greens and blues may look the same.

All these factors mean that screens will have to be fairly simple. Use large fonts whenever text is to be displayed on the screen. You should test all of your screens on a TV set to get an idea how everything will look. Don't have small items on the screen that are critical to operation of the application. This means that you shouldn't present too many options at once or make them too small. The "nine items on a screen" suggested limit was arrived at because when you try to put too many items on a TV screen, they begin to get very small and difficult to see. Try not to make options color dependent. When you ask the user to select the blue symbol, he may not be able to tell which one is blue.

IR Remote

While there will be a few users out there who will buy the optional keyboard and mice, Most users will be using the infrared remote controller exclusively. You should be sure to design your application so that people can use the IR device for just about everything. This will probably mean simplifying screens, limiting the number of options on each screen, and supplying defaults wherever possible. During product development it is probably a good idea to ignore the mouse entirely. See the "Remote Control and the User" section for a description of the remote controller.

The biggest hurdle to overcome when designing for the remote is positioning pointers. The remote is like a crude mouse, at best. Since it only registers in four pixel increments, fine positioning will be difficult. Wherever possible, you should design your application so that the user can cycle through the various options. Rather than saying "click on this" or "point and click at that", the user should be able to keep pressing the direction key on the IR remote until the option they desire is highlighted, and then they can press one of the select keys.

Users will not be able to drag items easily. They will not be able to move the cursor around the screen easily. Pointing at a small object on the screen will be difficult. Pull down menus, double-clicking, gadgets, etc., will either be impossible or difficult with the IR remote.

If you wish the user to make a choice between items or options, then build your program so that pressing the arrow keys (direction key on the IR remote) will cycle from one item to the next. "Point and click" operations translate into "cycle through options until the desired option is highlighted, then press the A key." You should also be very clear about which items are highlighted as the user cycles through them. Outline, frame, flash, reverse the background color, or animate the items as they are highlighted. The users must be able to see which items are highlighted from across the room.

You should try and make each of the options as clear as possible (or supply help screens or audio help). Symbols are better than text. Digitized symbols are better than hand-created symbols. Animated symbols are better than static symbols. If you find that you have to explain to someone what an symbol means, it would be a good idea to rethink them. If the purpose of an symbol is clear and intuitive to the user, then it saves them frustration. It saves you time and money spent on user instructions as well. It is important to keep in mind that as a developer of computer products you have come to accept certain symbols and actions as intuitive. This may not be the case with an average CDTV owner. A white rectangle with a folded corner may symbolize a test/document file to most computer literate people, but it doesn't mean anything to a noncomputer person. Test your symbols on non-computer people first.

When the user has an item highlighted and presses one of the select keys on the remote, it is a good idea to have an indication that the program got the message. A "busy", "working", audio beep, screen flash, fade to black, or other visual/auditory signal will prevent users from pressing the select key over and over or thinking that the machine is broken. Most appliances in the home give almost instant feedback. They either start right away or at least beep to indicate that they know a button has been pressed.

The Passive User

Even though a CDTV player is an interactive multi-media device, most people still would like to sit back and have things handed to them. Given a choice between reading all about a subject or watching a TV show that only skims over the information, most people will watch the TV. Unless the user feels that the goal is valuable to them, they won't invest much time or effort getting it. They also don't want to read a lot of instructions or spend too much time learning a new system. You have to determine how much involvement to expect from the users and hopefully come to a balance. The more that a user wants to get something from your product, the more he will be willing to read, learn, and interact with it. If the information or reward is great enough, people will go to great lengths to get it (just look at MS-DOS and how much effort it takes before someone reaps the benefits).

If the user feels only a casual interest, then the more obstacles you put in his way, the more frustrated the user will feel. In the case of the CDTV player an obstacle might be a long book of instructions, special keys to memorize, unique actions to perform, etc. Forcing the user to draw his chair closer to the screen in order to read or see something small is an obstacle. forcing him to do a lot of fine adjustments with the remote (such as fine positioning a pointer in a small area) is an obstacle. Perhaps the biggest obstacle would be requiring him to buy or have a keyboard or mouse.

Compare how difficult it is learning to drive a car with using a crepe maker. Most people will spend months learning to drive, stand in lines and take tests to get their license and spend thousands of dollars on cars because they see that the rewards are great. But even if you were given a crepe maker as a present, it ends up in a closet because it is just too much of an annoyance to make batter and clean the thing. If you feel that your particular application will be important enough to the user, you can make it as difficult as you like. If, on the other hand, you are offering them a crepe maker, then it better be self-cleaning.

You should try to build "time-out" features into your applications. In other words, if there is no input from the user after a few minutes, the program should revert to a self-running mode. This is particularly important at the beginning of a session. It would probably be a good idea to have your startup screens offer the choice of beginning right away or going through a tutorial. If nothing happens after a minute or two, jump right into the tutorial. Look at the way arcade game machines function. A mixture of self-running demos and instruction screens are constantly moving on the screen until someone puts in a quarter. People would much rather sit back and be shown how to use a thing (a few times if necessary) before they jump in rather than have to read printed instructions and then be dumped into something unfamiliar.

Don't be afraid to steal ideas from household appliances. How do people program a microwave oven? A stereo? A VCR? (Keep in mind that all over the world there are VCRs flashing 12:00...12:00...12:00 because setting the clock is just too much trouble or is too confusing.) How do people use a remote control device to watch TV? Do most people enter a channel number on the keypad, or isn't it just easier to press the up or down buttons a few times?

Summary: Design Considerations Unique to CDTV Applications.

Television (TV)

Televisions are not the same as computer RGB monitors. Television is an interlaced, overscanned medium. What looks good on a monitor, may look terrible on a TV set in the home. Some colors bleed, others do not stay "true". View your screens on a TV set before you commit them to CDTV disc. No matter how good your application is, if it doesn't look good on the home TV set then the users will be disappointed.

Use large fonts whenever text is to be displayed on the screen. Do not use small items on the screen that are critical to the operation of the application or present too many options at once. The suggested limit is nine items on a screen because when you put too many items on the TV, they get very small and difficult to see. Options that are color dependent are confusing and the user may not be able to tell which symbol is blue, preventing the proper selection. Test all of your screens on a TV to get an accurate impression of how it looks.

TV Imagery

Issues related to 6 to 8 foot viewing distance and TV resolution...

Fonts

Just about any font will work if it isn't too fancy in the first place. In completely unscientific tests anything below 20 point type becomes difficult to read from more than ten feet. Also, broadcast television character generators almost always use anti-aliased fonts with a neutral colored outline. Ideally, the outline color is halfway between the character color and the background color. Outlines, borders, and drop-shadows greatly improve readability. Pastel colored fonts work better than bright or primary colors. Off-white works better than pure white. Yellows, greys, and pale blues seem to work best.)

Fonts On Order. We are commissioning a few sets of fonts in various point sizes. They will be designed specifically for the CDTV support software and should be freely available to registered developers.

Colors

Colors should be subdued rather than bright (bright colors tend to "bleed" on poorly adjusted TV sets.) If you are using a paint program like Deluxe Paint from Electronic Arts that have color values in a range from 0 to 15 we strongly recommend that no value go above 12 or 13. The only safe way to test colors and color combinations for display on an NTSC or PAL TV is with a waveform monitor. Maximum values 85% IRE. The best advice is to keep the color values below saturation and look at the results on a television set.

Nationality and User Preferences Selection

While many people will adjust the Prefs (particularly nationality settings), most people will never touch them (if it ain't broke don't fix it). However, certain adjustments will have to be made by the user upon power up or after a loss of power. We will try to make the transition between Preferences selection and normal use as seamless as possible.

Television Sizes

Some people will be using small TV sets. While we might have colored borders around the edges of screen developers should work within a “safe” area.

Appendix C: IR Remote Control Description

Refer to the Developer Notes for a complete description of the key codes generated by the Remote Control.

On the left side of the remote is an eight-way direction key, which is used to move pointers on the screen, make selections, etc. There is another button on the remote for toggling how the remote direction key operates, either as a joystick or a mouse. When in the joystick mode, the direction and fire button signals sent to the player will be in standard joy, four button, eight-way form. When the remote is in the mouse mode, intuition mouse movement calls will return values in four pixel increments rather than normal Amiga one-pixel increments. This was done, primarily to enhance the responsiveness and speed when moving a pointer. Developers should try and write their applications to trap for both modes if possible and notify the user to press the joy/mouse button to change modes if necessary. A simple "press left selection button to start" message to the user will let you determine what mode the remote is in.

On the right side of the remote are left and right selector keys which are the equivalent of the left and right mouse buttons in mouse mode. In joystick mode, only the left selector button will function as a fire button.

Next to the selector keys are two buttons for adjusting the headphone volume up and down and a power button for turning the player on and off. Next to the volume controls are buttons for controlling all the audio functions of the CDTV player (rewind, fast-forward, play/pause, and stop).

Above the audio control buttons are three buttons, first a genlock button for use with the optional genlocking accessory. This button will cycle the CDTV player through the three genlock modes: source only, mixed source and computer, and computer only. Next is a button for switching between CDTV and normal TV viewing (this toggles the RF modulator and light on the player front panel). Finally is the joystick/mouse toggle button.

To the right of the direction key are 12 buttons (10 buttons numbered zero through nine, an escape button, and an enter button). They will be laid out in the same fashion as a telephone keypad excepting that the escape button is to the right of key 3, the zero button to the right of the 6, and the enter button to the right of the 9. These keys will send the same values as those found on the Amiga numeric keypad.

As a positioning device any remote unit will be awkward to use for fine adjustments (even a mouse is not completely intuitive). For that reason we should not have gadgets for resizing, dragging, etc.

To avoid forcing the users to precisely position a pointer (and going along with some of the TV specific imagery rules) we felt it would be better to cycle through any list of selectable options.

We have established a set definition of any key functions (such as help or return to main screen). More thinking is required on how a user will use the IR device for scrolling through lists, controlling text speed and size, etc. There may come a time when an application requires some form of text entry and not everyone will own a keyboard.

The Escape button's function in all cases is to bring up some kind of help screen. This help screen could be as simple as an symbol telling the user to put in the welcome disc (this would be the only thing handled by ROM.) Developers could have the Escape button activate a generic help screen of their own, give the user the option of going through a demo or tutorial section, refer the user to their manual, bring up contextual help screens, or have special options such as "return to top", "return to previous screen", "jump to another section", "jump to another mode", etc.

The Return button is used to make selections or activate an option in just about every case. In some cases (title dependent) the select or A/B buttons might perform this function for example if a title is Amiga mouse dependent (which we do not encourage.)

The arrow buttons are always used to move around. Move from symbol to symbol, move through a list, move to another selection, etc.

Designing Screens for CDTV Multimedia

Introduction

The success of a CDTV title depends in large part on the quality of its user interface. Attractive, easy-to-use applications are well-received, often demonstrated, and generally have higher retail sales. For this reason, it is important for CDTV designers and artists to understand the fundamentals of screen design and layout.

This article discusses some of the basics of screen design. None of the following suggestions is sacred, as applications often dictate special conditions. Any particular application, *with good reason*, can easily violate any or all of these guidelines. For example, some of the best computer games contradict design suggestions in this article because the game play itself is unique and some of the guidelines are inappropriate for the game play. However, good reasons for breaking these rules do not include poor development practices such as lack of sufficient alpha or beta testing, incompetent programming, or amateurish graphic artwork.

By understanding the following guidelines, developers will have a framework within which they can decide how to bend the rules, if necessary, without harm.

Designing Screens

There are at least two major factors to take into account when designing graphic screens for multimedia applications: Usability and Aesthetics.

Usability

Usability refers to how easy or intuitive the screens are to operate. Users should be able to quickly figure out what they are supposed to do, and how to do it.

Aesthetics

The aesthetics of screen design refers the screen's visual appeal, in terms of eye pleasing, artistic qualities appropriate to the type of application.

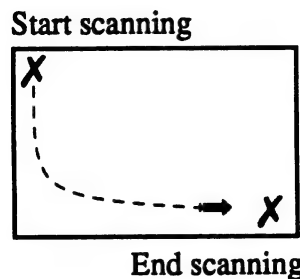
While usability and aesthetics can be considered separately, they are interrelated. A screen that may initially please the eye will not do so for long if the user gets frustrated trying to figure out how to operate it. Likewise, a screen that is not visually appealing will be unimpressive, no matter how easy it is to use. This article looks at four broad areas: Screen Layout, Color and Motion, Text Readability, and Consistency.

Screen Layout

The layout of a computer graphic screen has a great impact on ease of use and visual appeal. Several general rules should be kept in mind.

Scanning Patterns

When users first see a new screen they usually scan it quickly in a very definite pattern to get acquainted with it. Normally the user starts at the upper left corner of the screen, scans quickly down the screen vertically, and ends at the lower right corner. Scanning helps one get visually oriented before settling into reading text or performing interactions. This is similar to scanning a printed page for paragraphs and headings before actually starting to read the text. It follows the normal top-down, left-to-right reading pattern on a printed page that is common to all Western languages.



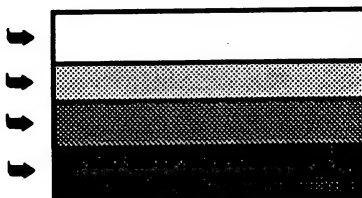
Screen designers can exploit this scanning pattern to facilitate the user's interaction with the screen. For example, since the upper left corner is the user's starting place, it is a good place to put important orienting information or instructions (such as a screen title or a prompt for user action). The lower right corner of the screen is a good place to put the most likely final action for the screen, such as a "Continue," "Next," or "Return" button because it is the last place a user is likely to scan.

Choose One:

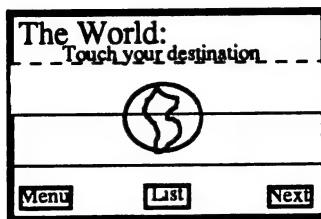
①	②
③	④

Zones and Regions

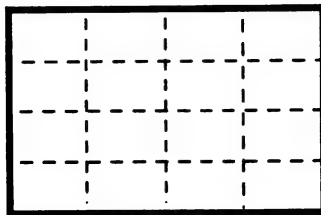
Between the screen's upper left and lower right corners is the space for user interaction. Users tend to scan down the screen vertically to get an idea how the screen is organized, so it is helpful if the screen is broken up into two or more vertically arranged "zones," extending horizontally across the screen. These zones can be thought of as functional regions where related choices can be made or where text instructions can be placed.

Zones:

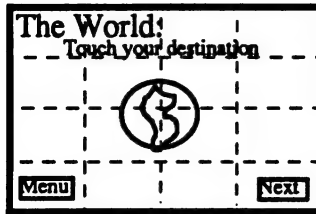
Organizing the screen in horizontal zones, both visually and functionally, helps the user to quickly locate the desired area of the screen. After locating the appropriate zone, the user can then easily read across the zone and focus on the details of information or interaction inside the zone. For example, a screen title placed in the upper quarter of the screen can be thought of as occupying a functional zone. Likewise, a row of navigational buttons across the bottom of the screen comprise a zone where related functions (Main Menu, Help, Return, etc.) are easily found.

**The Grid**

The grid is a basic tool for all graphic design because it is indispensable for arranging and balancing graphical elements on the screen. These elements include text, images, and interactive buttons. Grids can be of any reasonable dimension, most commonly 3 x 3, 4 x 4, and 5 x 5, as well as combinations such as 3 x 4, etc.



Not only do grids contribute to the visual balance of a particular screen, but using a common grid for all screens in an application maintains a consistent visual style across the entire application. Such consistency of style is well-appreciated by users. By designating some areas in the grid for interactive functionality and balancing them against "non-interactive" elements such as images, titles, or text, a pleasing, flexible arrangement can be created and maintained (with modifications) across an entire series of screens.



Some developers and programmers may initially object to using a grid because it seems inflexible. However, professional graphic artists regularly use grids because they actually increase design flexibility. Grids provide the artist with a structure to follow for consistency, and a structure to diverge from ("play off") for emphasis and artistic license. The grid is a very valuable tool for creating a well-balanced, visually pleasing layout.

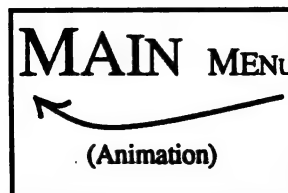
Attracting the Eye with Motion and Color

The user's eye will be attracted to different areas of the screen in varying degrees, depending on factors such as motion and color.

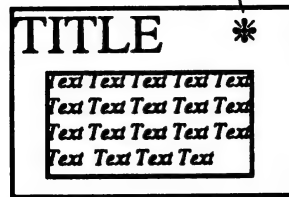
Motion

The user's eye is most strongly attracted to any motion on the screen. This is a physical/biological response of the human eye to movement and is often exploited for advertising purposes in television commercials, moving signs, etc. A multimedia designer can use this reaction to movement to draw the user's attention to a specific area of the screen.

For example, a new screen might be introduced by a spinning title that lands in the upper quarter of the screen. This will not only give the screen visual appeal, but will also draw attention to the title, thereby helping to orient the user. Another example is a "Return" button that starts blinking after a timeout period to attract the user's attention to the logical button, thus guiding her through the application.



On the other hand, unnecessary or excessive motion on the screen can be very distracting. If a text screen has a decorative animation playing in one corner, the text will be harder to read because the user's eye will be continually distracted away from the text.

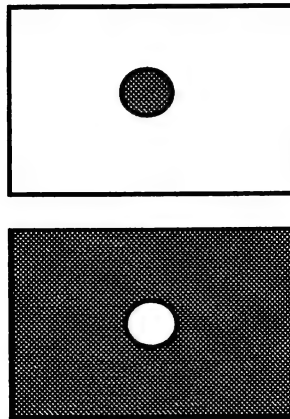
Incorrect use of Animation

Text that is moving is also more difficult to read. If you want to call the user's attention to some text, it is usually better to flash a border around it for emphasis. If the text is very short (a few words) you might blink it very slowly with plenty of time between blinks to read it.

Color

Color is another major attraction for the eye. In general, the hotter colors (reds, yellows, pinks etc.) attract the eye and tend to come forward on the screen, while the cooler colors (blues, greens, browns, etc.) do not attract the eye and tend to recede.

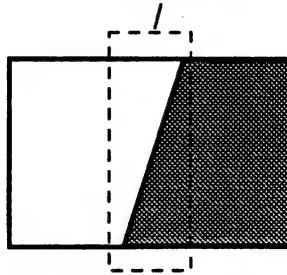
Consider a dark gray circle on a light background. This will most likely be perceived as a hole or depression. A light yellow circle on a dark background, however, will most likely be seen as an object in the foreground over a dark background.



In addition, the lighter and more highly saturated the color, the more strongly it will attract the user's eye. Therefore while blue is not naturally a hot color, a brilliant blue (light and highly saturated) can overpower a muted red (dark, low saturation). If the brilliant blue is placed as text on the muted red background, the viewer's eye may feel a tension and fighting between the colors on the screen. This is because a receding color (blue) is being placed on top of a color (red) that naturally tends to come forward.

The area of strongest attraction for the user's eye is normally the area of highest contrast on the screen. This occurs at the borders between the lightest and darkest areas (or hottest and coolest colors) where the difference is greatest. For example, if there are only two colors on the screen, say a rich lemon yellow (hot) and a dark grayish blue (cool), the border area or edge where they touch will attract the user's eye. To attract the user's attention to an area of the screen, the designer could use a small box containing stripes or speckles of yellow on the blue. This will provide an area of high contrast between the two colors to attract the user's eye.

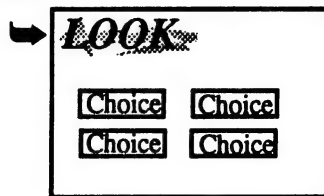
Area of strongest attraction.



Background and Foreground colors

When choosing colors for screen display, it is important to consider the foreground and background colors carefully. Background colors should, in general, be cool, darker, less-saturated colors that recede and do not vie for the user's attention. These cooler background-type colors include olive greens, grays, blues, browns, dark purples, and black. Conversely, foreground colors can be hotter, lighter, and more highly-saturated colors (lemon yellow, pink, orange, red) that tend to come forward on the screen and attract the user's eye.

Contrast area



Color Coordination

The visual appeal of a screen is determined in large part by the colorfulness of the screen. However, this does not mean that the more colors the better! *The key to using color effectively is using it conservatively.* In fact, the most visually appealing screens often use only two or three hues (pure colors such as red, orange, yellow, green, blue or purple) and vary the shades (brightness) of those hues. In addition, black, white and gray can always be used.

For example, a color combination of yellow and blue can be very attractive if the designer uses two or three shades of blue, two or three shades of yellow, and black, gray, and white. This gives the artist a good selection to choose from, yet maintains a visually pleasing color harmony. In this way, the screen's color coordination can be established and then creatively modified.

If, on the other hand, the same screen were designed using six different hues (green, blue, yellow, orange, red, purple), the confusing mixture would likely distract and repel the user.

Color Blindness

Multimedia screen designers should remember that an estimated 8% of the adult male population is red/green color blind. Therefore avoid red/green color combinations such as red text on a green background (or vice versa), especially when the reds and greens involved have similar luminance (brightness) values.

One technique that can help designers select the proper luminance of different items on the screen is to design screens in black, white and gray (to test the readability and overall design), and add colors later while maintaining the original luminance values.

Text Readability

The readability of text is of primary concern to the multimedia screen designer. Easily readable text is essential to a successful application. Text that is hard to read will not only annoy and frustrate the user, it will also lessen the overall impact of the title. Some basic elements of readability follow.

All Upper Case vs. Upper and Lower Case

Words in upper and lower case are more easily read than those in all upper case. All upper case characters should be used occasionally, and only then for the purposes of emphasis.

THE REASON UPPER AND LOWER CASE TEXT IS EASIER TO READ THAN ALL UPPER CASTE TEXT IS THAT MOST PEOPLE QUICKLY RECOGNIZE WORDS BY SHAPE, NOT BY READING EACH LETTER. THEREFORE, SINCE LOWER CASE LETTERS CONTAIN ASCENDERS AND DESCENDERS, THERE IS MORE VARIATION IN THE WORD SHAPES, AND THUS THEY ARE EASIER TO RECOGNIZE.

The reason upper and lower case text is easier to read than all upper caste text is that most people quickly recognize words by shape, not by reading each letter. Therefore, since lower case letters contain ascenders and descenders, there is more variation in the word shapes, and thus they are easier to recognize.

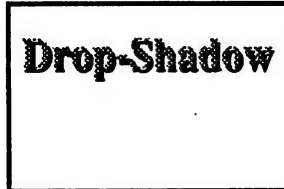
Flicker

Since CDTV applications are to be viewed on an interlaced display (television), the designer must take into account a phenomenon known as "Interlacing flicker," or commonly just "flicker." Flickering characters are seen to "shake" on the screen, and this is highly objectionable to a user trying to read text. Flicker is most noticeable on horizontal lines of single pixel height between two sharply contrasting colors (e.g., a bright white line on a red background). Notice the flicker on the weather maps of local television news programs. Often the smaller characters and thinner lines on these weather maps seem to "shimmer" or "shake" over the map's background. Sometimes flicker is most noticeable in an area of the screen in your peripheral vision, that is, if you are not looking at it directly. In addition, flicker is considerably more apparent in PAL mode, since PAL runs at a slower screen refresh rate (50 Hz, as opposed to 60 Hz for NTSC).

Flicker can be dealt with by at least three methods: anti-aliasing, drop-shadows, and avoiding single-pixel high lines. Anti-aliasing helps smooth out the flickering of the high-contrast edges between colors. Drop-shadows help pull the characters out away from the background, and give a hard black edge against which to better recognize text on the screen. Avoiding single-pixel high lines in contrasting colors removes the problem of flicker altogether.

Anti-aliasing is the process of inserting transitional colors between two highly contrasting colors (e.g., a middle gray pixel between a white area and a black area, or a medium brown between a lemon yellow and a dark purple.) This transitional color minimizes the apparent flicker on the screen and greatly reduces eyestrain. Anti-aliasing helps smooth out the flickering of the high contrast edges between colors. How anti-aliasing is accomplished varies depending on the software tools. With some graphics programs fonts and brushes can be automatically anti-aliased, saving many hours of manual retouching.

Drop shadows can usually be created by picking up the text as a brush, stamping it down in black or gray to create the shadow first, then changing the brush back to its original color and stamping it back down on top of the shadow, but slightly offset. Drop shadows help pull the characters out away from the background, and give a hard black edge against which the user can better recognize the characters and words.



To avoid single-pixel high horizontal lines, the simplest method is to use low-res graphics (200 vertical lines of resolution). This resolution mode may not be of sufficient quality for a particular graphic image. If the graphic is high-res (400 lines of resolution), the single-pixel high lines can be retouched by hand.

Color Choices

Choosing the right colors for the user's environment is very important to the look and usability of a CDTV title. There are significant differences in the way computer monitors and color televisions display colors. Images that look crisp and clean on a computer monitor can easily become "messy," muted, or blurred on a television. It is important, therefore to tone down the hottest colors (especially red), and to look at your screens often during the design process on a home color television or a composite input.

For example, a bright red might look very good on the designer's computer monitor, but on a television, that same red could be too hot, and bleed or smear red to the right.

Another design consideration is whether the program will be running in Europe on a PAL system, in North America in NTSC, or both. PAL and NTSC differ in their luminance values; NTSC tends to display screens significantly brighter (about four shades brighter). An application created on a PAL system may display colors much differently on an NTSC system. This is especially significant when it comes to background colors, which may appear to be too bright in NTSC, or even a totally different color. A nice rust-brown background in PAL, for example, may show up as an intolerably light yellow in NTSC.

Consistency

Consistency refers to a common look and feel across various screens within a single application or a series of applications.

Necessity of Principle

Consistency, as a principle, is extremely important when designing multimedia screens, for both aesthetic and functional reasons. Aesthetically, a multimedia application that has a consistent design in terms of colors, style, and layout will be seen as a much higher quality program than one that uses a variety of backgrounds, colors, and layouts across multiple screens. Functionally, a consistent design helps the user to quickly learn to operate the user interface and navigate through the application.

A program that maintains a tight consistency gives the user a feeling of unity and overall vision of design, which lends a valuable sense of credibility and integrity to the application. An inconsistent look within the application forces users to continually reorient themselves at each screen, and thereby distracts them from the subject or content of the program. Consistency also cuts down drastically on the time spent developing screens, since the common elements, such as backgrounds and buttons, do not have to be continually created.

Consistency can be applied to the look of an application through colors, button design (3-D, drop shadows, etc.), button placement (bottom row, screen corners, etc.) background textures (paneling, stucco, tiled, etc.) and so on. It is worthwhile to take some time and experiment with various graphic styles, decide on one style and then stick with it instead of continually redesigning and laying out each screen anew.

The other aspect of consistency is the method(s) of interaction (scrolling list arrows, on-screen keyboards, Return buttons, etc.). By being consistent in how the user interacts with the program from screen to screen, the developer allows the user to move through an application much faster. This especially applies to items such as navigation buttons. If these items (Main menu, Return, Help, etc.) are inconsistent, the user's progress through the program is impeded. Users become accustomed to button locations, methods of interaction, etc., and if these change within an application it will confuse the user, slow him down, and create frustration.

When to Break the Rules

The number one priority of an application is ease of use. All else is sacrificed on this altar including consistency of design. When adhering to a particular consistency of screen design makes an application more difficult to use, it is time to break the design rules in favor of ease of use.

Imagine an application in which the lower right corner of the screen contains a button labeled "Next," to advance the user to the next screen in a sequence. The application is designed to blink the button after thirty seconds to indicate that it is waiting to be pressed. However, on one of the screens the user is supposed to make a choice before moving on to the next screen. If the thirty second rule is blindly adhered to in order to be consistent, the "Next" button may start blinking before the user has a chance to consider the choices. This could result in steering her into an incorrect action (pressing the "Next" button rather than one of the choices). In this case, the thirty second rule for the "Next" button should be broken in favor of blinking the instructions to the user about making a choice. Once the user has made a choice, the thirty rule for the "Next" button can be reinstated.

Ironclad adherence to consistency is an extremely common mistake. Remember, the most important consideration is ease of use, not consistency of design (and certainly not ease of programming).

Conclusion

Good screen design and layout contributes to the overall success of a CDTV title. Visually appealing and easy to operate screens are well received by users and subsequently by all those involved in marketing and selling the product.

However, good screen design is not arrived at by chance or good intentions alone. It is only possible if the designer understands the natural tendencies of users to scan screens, and react to color, movement, and layout, and if the designer makes use of these natural tendencies to help the user through the application.

If your testing results or reviews indicate that a user has trouble understanding or operating a screen, check your screen design for the basic principles of layout and interaction. If you design with the users in mind, they will surely enjoy using your application and will recommend it to other CDTV owners.

Localizing CDTV and CDTV Applications

Consumer products are a good example of localized products. The car you drive, unless you made a special purchase arrangement, is localized to your measurement system and driving standard. The same holds for electronic components like stereos and televisions. From the labels on the front to the power supply in the back, the unit is ready out of the box for use in your country.

Localization for computer products used to be an option, now it is a necessity. This is true for Commodore because the majority of Amiga sales are in Europe and especially true for CDTV because it is a consumer product more than it is a computer product. Initial CDTV sales are expected to follow the same European sales pattern as Commodore's computers and so CDTV has been designed with localization in mind.

All CDTV units contain both PAL and NTSC crystals, and a universal power supply. The power supply is capable of detecting whether 110v or 240v, and 50Hz or 60Hz is required. PAL and NTSC selection, however, can only be done by cutting or connecting jumpers on the motherboard. Future revisions of CDTV will include a hardware switch on the outside of the machine to select PAL or NTSC.

CDTV system software is localized by user selection of settings on the Preferences screen. Applications should then present all messages in a manner consistent with the Preferences settings. This is the runtime processing for localization.

CDTV applications are localized at development time by translating all application specific text and documentation into the user's language. In addition, CDTV applications offer unique localization challenges due to the inclusion of speech with most applications and the predominance of symbols over text.

CDTV Preferences

The CDTV Preferences screen allows the user to set two Preferences items dealing with localization:

Language

Currently, the user has a choice of fifteen languages

CDTV Preferences Language Choices

American English	English	German	French
Spanish	Italian	Portuguese	Danish
Dutch	Norwegian	Finnish	Swedish
Japanese	Chinese	Korean	

Time

Either AM/PM or 24 hour.

CDTV applications should check the user's Preferences settings and change all relevant features to support those settings.

Translation

Translation for CDTV applications covers four areas:

1. The application screens.
2. The spoken portion of the application.
3. Cultural issues and symbols associated with the application.
4. The application documentation.

One issue of translation is that English text tends to expand when translated into European languages. The increase can be as much as 30–50%. Naturally, the reverse is true. Whatever your original language, you should make provisions for this when coding your application.

Another issue is who does the translation. You should use a translation company experienced in translating technical material.

Applications like dictionaries and encyclopedias should not even be translated. It's far easier (and cheaper) to license an existing dictionary or encyclopedia already produced in a country than to attempt translation of a 21-volume encyclopedia.

The Application Screens.

Most CDTV screens should have minimal text. This is beneficial in two ways.

- The first is that CDTV applications should be intuitive and text isn't intuitive; pictures and symbols are.
- The second, and most important from a localization standpoint, is that less text means less translation.

Any text on an application screen is subject to translation. This includes dates, times and numeric values. For example, fractional values are expressed differently in Europe than in the United States. Your application should present all relevant quantities in the correct format for the user.

Screen layout can be adversely affected by the translation process. Do not code your application's screens to rely so heavily on text being a particular length that the deletion or addition of words will ruin the look of the screen.

The Spoken Portion Of The Application.

This is a crucial part of any translated application. The large capacity of a CD allows the CDTV developer to include spoken text and music that can greatly enhance an application. However, that enhancement quickly becomes a detraction if poorly done.

Spoken text translation has two elements—the translation of the text itself and the manner in which it is spoken. When the text is translated, you must not only use the correct phrases for that particular language, you must also use the correct phrases for the CDTV technology. Translation is not always word for word. In fact, some words may not even be translated.

How the speaker sounds is as important as what is said. The speaker of the translated text must have the correct accent for that language. You don't want the German text in your application to be spoken with a Japanese accent. Also, don't use the standard Amiga *narrator.device*—its phonemes

are not suitable for international speech. The best approach is to use a professionally trained narrator or speaker who is fluent in the target language.

Avoid long sentences which get even longer when translated. Use short phrases which lend themselves well to translation. Another thing to avoid is tight time synchronization between displays and spoken text. The expansion of translated text will most likely throw off the synchronization and you will have a lot of recoding to do.

There are translation service centers which do varying degrees of translation from translating written text to finding an actor to speak the text and sending you a tape to even sending the entire translated text and narrative back to you on a SCSI disk.

Cultural Issues Associated With The Application.

Cultural references and symbols that are acceptable here, may not be in other countries. For example, instructing children to "look left before stepping into the street" is fine for the United States, but dangerous for the United Kingdom.

The symbols used in an translated application should make sense for the country involved. The mailbox symbol people are accustomed to in the United States—a box with a rounded top and a small door—may mean nothing to a European or a Canadian. Use symbols appropriate to the country. The same holds for place names on maps. Even the meaning of colors, such as amber for warning in the United States, is subject to cultural differences.

A good translation company will go over cultural issues with you in addition to translating your text.

The Application Documentation.

The entire text of the application's documentation will be translated into the target language(s). The same factors involved in coding the application hold for the documentation. Fortunately, CDTV applications do not have a lot of documentation to begin with, so this will not be an extensive process.

PAL vs. NTSC

PAL screens have 256 scan lines non-interlaced and 512 scan lines interlaced as compared to 200 and 400, respectively, for NTSC. It is important that applications be tested under both standards to determine if changes are required to the screen layouts to make them compatible.

Color rendering is different between PAL and NTSC. Avoid using intense, highly saturated colors. If possible, test your NTSC applications on a composite PAL monitor and TV set, and vice versa.

CDTV Glossary

The CDTV glossary that appears in the CDTV User Interface Guidelines lists CDTV and CDTV-related terms and their definitions. Some of the definitions include the recommended usage of the term in CDTV applications. For example, the definition for external device states that it is "something attached to a CDTV player" and recommends that it be referred to an "accessory".

The glossary has been translated into a number of languages and is available from Commodore.

[REDACTED]



CDTV Title Guidelines

A CDTV title is not simply an Amiga application running in a different box. The CDTV player imposes certain restrictions on a title—no menus and large icons, for example, and provides certain benefits—large storage capacity and digital audio. The wise CDTV developer respects the former and takes advantage of the latter.

The list below gives you, the developer, a quick reference to the do's and don'ts of CDTV titles. It contains rules and common sense advice. They are broken into two groups, minimum requirements and quality standards.

Minimum Requirements

The minimum necessary to be an acceptable CDTV title.

Quality Standards

To get into people's homes, you need to do more than the minimum. These will help you make the trip.

LEVEL 1 QUALITY: MINIMUM REQUIREMENTS

1. No program crashes. The title should not crash, guru or otherwise cease to be functional. Test, retest and test again till you are sure your title is robust.
2. No logic or flow errors. The title cannot take a path other than the one requested or expected by the user. For example, if the user asks for a map, but instead gets a picture of a tree, a logic or flow error has occurred.
3. All images presented should be free of error and look clean. For example, a title should not have a garbled picture or a video sequence that exhibits solarization, i.e., a color picture that looks like a negative.
4. No low quality images. All still images should be high quality, preferably digitized interlaced HAM images. Drawings or animations should be detailed and free of major color banding. All still images should be overscanned unless a conscious effort is made to provide a colored border.
5. User interface. The program should follow generally accepted CDTV interface rules including:
 - a) A button for action, B button for backup, arrow keys move in direction of arrow.
 - b) Single click to select an object.
 - c) Use highlighted hitboxes rather than a pointer where possible.
 - d) Highlighted hitboxes should be accessible by cursor keys in any direction.
 - e) If a pointer is used for products with invisible hot boxes or for special purposes such as coloring, the pointer should change when it is over an invisible hot box and be in a form relevant to the title (paint brush, wand, etc.).
 - f) Numbered items should allow use of the numeric keypad on the controller.

- g) Selectable items should stand out (e.g., 3D buttons) from non-selectable items, and they should give audio/visual feedback when selected.
 - h) Selectable items should give appropriate, consistent, and predictable results.
 - i) There should be no references to a computer keyboard (e.g., F1 key).
6. The title should look good on any television. This means you should buy a *cheap* television for testing.
 7. There should be no signs of AmigaDOS. Examples include the AmigaDOS cursor, Workbench screen, system requesters, sleep icon, pull down menus, flashing title bar, front/back gadgets, or jargon (x memory free, loading next module, etc.).
 8. Efforts must be made to reduce perceived boot-up time. The titlescreen should appear within five seconds of the appearance of the CDTV Interactive Multimedia logo. (See Discis' products) The program should show a title screen before doing anything else. It should not show CLI, Workbench, or any pointer.
 9. It must have a screen blanker tied to preferences. We recommend the screen blanker supplied as part of the OS.
 10. Titles must work under AmigaDOS 1.3 and 2.0 in both NTSC and PAL. Programs should be able to successfully pass *enforcer* and *mungwall* testing.
 11. The program must be designed for use on a PAL or NTSC TV, which means care must be taken in regard to all graphic elements (fonts, symbols, pictures, animations, video) with respect to size, style, color combinations, and contrast. Test your titles on those two environments, not just with a monitor and one of the two standards. Specific suggestions include:
 - a) Fonts should be simple with no thin lines, anti-aliased, easy to read on a television and at least 20 point size.
 - b) Text should generally be highly contrasted to its background.
 - c) Text should have borders or drop shadows to make it more readable.
 - d) Don't use pure colors (R, G, B values should be less than or equal to 13 out of a range of 0-15) because they bleed on television sets.
 - e) Be careful of the colors used as some colors show up very differently on NTSC versus PAL. For example, deep red in NTSC comes out pale pink in PAL. *The only way to find this out is to test on both systems.*
 - f) Avoid stark contrasts when using thin horizontal lines since this will not look good in an interlaced medium (TV), and avoid single pixel horizontal lines entirely.
 - g) Do not base instructions solely on color, i.e., don't state "Pick the orange button" since TV sets will be adjusted differently. This could also be a problem for colorblind users.
 - h) There should be no more than nine selectable (by cursor or by pointer) items on a screen unless the individual items are recognizable because they are part of a set (i.e., alphabet, numbers, states). Nine items fit well with the font size required for television.
 12. Products must not substitute repetitiveness for depth by reusing the same elements in different places. If a product is perceptually redundant, it is boring. For example, using a passage from Beethoven's Piano Concerto No. 5 as an example of his music, and as an example of how a piano sounds, and as an example of a piano concerto is a lack of depth.
 13. Eliminate all spelling and grammatical errors; people will not want to use a product, especially an education product, if they cannot trust something elementary like its spelling. Run your

text through a spell checker and a grammar checker. Some of these titles are available in UK English or American English only, and these are acceptable, at least for the initial shipment.

14. Programs should reboot when the disc is removed unless the program disc needs to be removed for the product to be usable (*CD-Remix*). The program should reboot when the eject button is pushed, and the reboot should occur even if the disk is being accessed or Amiga audio is playing.
15. Sound quality should match the title requirement. Use Amiga sounds for audio feedback; CD-DA for game background, dramatic intro music and other sections designed to evoke an emotional response. All sounds should be clear and free from hiss or other extraneous problems. Speech must be ungarbled and unclipped and digitized at a reasonable level or be CD-DA.
16. Volume levels of speech, music, and sound effects should be uniform throughout the product. All audio must come through both channels unless there is a compelling reason to do otherwise. Note that compelling *does not* mean being unwilling to take the time to code so that the sounds comes through both channels nor does it mean that your authoring system only works with one channel. Compelling *does* mean trying to add depth to the sound by having one person come through the right channel and another through the left channel.
17. Interruptability. All titles need to be interruptable at any time, including title and credit screens, introduction, during accesses, or animations.
18. Products must use preferences for language selections. Unless the language chosen in preferences is unavailable, the user should not normally see language selection screens.
19. All programs that can save to a floppy must be able to format a disk.
20. All programs should test for joystick/mouse mode. If the controller is not in the proper mode, it should ask the user to change modes.
21. Programs should disable keys that are not functional in the product. Typically this means disabling the audio keys for CD control.
22. Controller responsiveness. The product should not queue up button presses, it should react and give feedback immediately, and any cursor or highlight should move quickly enough for that specific title. In many cases, if a pointer is used it should include an accelerator feature. If a user feels compelled to repeat an operation because there is no response, the title is at fault.
23. The products should not have *any* dead time, i.e., time when nothing is occurring. Accesses should first give audio and visual feedback that a selection has been made, then have a transition of some sort, then begin the load during the transition. The transition interlude can consist of music, color cycling, a voice over, a fade to a colored screen, or in some way distract the user. A sleep or load symbol is generally insufficient to improve the perception.
24. Test that your product works properly with a trackball and a mouse.
25. They should also not be adversely affected by the presence of video peripherals such as genlocks.

Reference Titles

The reason someone purchases or uses a reference title is for the information contained within. A reference title should not have any of the following:

26. **Inaccurate reference data.** Imagine you're a student doing a homework assignment, using the CDTV title as a reference work. Your teacher gives you an "F" because your facts are wrong.
27. **Missing information.** If a menu, icon or other reference indicates that information relating to the subject matter is available, the information should be accessible from that point. In other words, if something is selectable, it must present the data associated with it.
28. **An inability to accept keyboard input, print, or save to disk even though most people will not be able to take advantage of these features at the moment.**

Recreation Titles

29. **A title must be playable to completion.** No user or program error should prohibit the game from continuing. If you make a stupid move and get eaten by a dragon and the game ends, you have played to completion. If you make an incorrect move and the game freezes up or prohibits the continuation of play, it is a not move that shouldn't have been made, it is a bug.
30. **A multiple player option should be in every recreational product.** Where it makes sense (certain sports and arcade games), two-player simultaneous play is a requirement (e.g., hockey and football).
31. **Simulations must attempt to match the real world in as much detail as possible, including the standard rules of play in sports games.**

LEVEL 2 QUALITY: THE NEXT STANDARD

In addition to the requirements of the Level One, products need to be compelling enough to compete successfully in the marketplace.

32. **All titles must have an important and distinguishing value over doing the product on magnetic media, or by book, or by cassette.** Products should have greater detail, more choices, more "sizzle", be easier to use, or be faster to perform a function. Ports from another platform—including the Amiga—must be enhanced (music, speech, additional video, more choices, etc.). An example of an excellent port is *SimCity* which added digital audio and rewrote the user interface to take advantage of the numeric keypad on the IR controller.
33. **Timely response is important.**
 - a) **On a multitasking operating system, the time that elapses from when a selection is made till the activity begins should be no more than three seconds.** This is part perception (i.e., start showing a graphic change while still loading), part disk organization (to speed access times), and part programming (sometimes things can be cached or optimized). (*Asterix* appears to have achieved this goal, so it is therefore possible.) To reiterate, first audio/visual feedback, then some type of transition interlude which lasts no longer than three seconds, then the desired result.

- b) For very long searches that cannot be done in a short period of time, inform the user of the progress of the search. Options include putting up a screen and start listing "hits" or showing a "gas gauge" depicting the progress of a search. The user should be able to halt a long search in progress, retaining the results found to that point.
- 34. Multimedia elements should be comparable to video or cartoons viewed on TV. These elements (animations, speech, music, sounds, video) should be streamed from disk so that they can be more in-depth and longer in duration. The animations should normally be 3 dimensional and change focus (i.e., background, perspective), not limited to a static background screen.
- 35. Educational titles and adventure type recreational products need to have a depth of interactivity options. For instance, if a character is walking down a street, the user should be able to go down alleyways, into buildings, etc. Each screen or in each section should have more than one (and more than two!) things that can be done. These options should include non-linear choices, i.e., being able to jump around. Linear choices are really no choices at all because you must follow a prescribed path.
- 36. Educational titles should have some type of testing function to allow you to examine your progress in a section. The Bookmark feature should be used if appropriate (e.g., game scores, place in a book, tests, etc.).
- 37. Reference titles should allow numbers and spaces to be input for searches. All reference titles should support searches on keywords in body or title, and not be just an alphabetized index of options (similar to the index of a book). They should also have the Bookmark feature using Non-Volatile RAM (NVR) to save search criteria and possibly the resultant elements.
- 38. Recreational titles should use continuous streamed animations and CD audio for background. They should be able to save game states and high scores using NVR.
- 39. Possible suggestions:
 - a) Online help
 - b) Templates to fit on top of the IR controller to simplify the buttons for complex products (e.g., flight simulator).
 - c) Optionally viewable demo commercials of other products.
 - d) Hardware add-ons (a la Nintendo).
 - e) Supply a formatted disk (or at least a disk label) if the product can use a floppy.

Programming CDTV

CDTV is not just another Amiga. While you've heard this before, it must be the starting point in programming a CDTV application (as differentiated from a port of an Amiga application to CDTV).

CDTV is certainly based on an Amiga—the A500, but CDTV has certain additional hardware which you must account for; it has certain user interface standards differences, which you should adhere to, and the typical CDTV user will probably be different than the typical A500 user.

Major Hardware Differences Between CDTV and A500

1. CD-ROM Drive
2. 1 Mbyte of Chip RAM
3. No Internal Floppy Drive
4. Non-volatile RAM
5. 'Credit card' RAM and/or ROM expansion capability
6. Infrared controller instead of a keyboard and mouse

These hardware differences shape what CDTV can do than a standard A500 cannot...and what an A500 can do that the standard CDTV cannot. It is important to note the characteristics of each of these hardware differences, so the application is well adapted to the CDTV environment. The hardware differences have major implications for the design of CDTV application software.

The CD-ROM drive can be considered as a large capacity storage device with slow seek times and a relatively slow data transfer rate. The large capacity makes CDTV a good vehicle for data intensive application software packages. However, the speed issues must be addressed, and of the two problems, the slow seek times will affect the normal application much more than the transfer rates.

While the transfer rate is relatively slow compared to that of a modern hard disk and controller, its speed (150K/sec) is adequate for most tasks. The time it takes your application to locate the data on the disk will be much more significant, as it is possible for the time it takes for the CD-ROM drive to seek from one place on the disk to another to be measured in *seconds*. This kind of delay can seriously affect your application's performance in the eyes of the user. Every effort must be made to minimize the distance that your application needs to seek as well as to hide the seek time from the user.

The first line of defense from slow seek times is to arrange the data properly on the CD disc. If the data can be arranged so that the head does not have to seek much as the user proceeds through the application, it may be possible to entirely hide the slow seeks. In some cases it may make sense to duplicate the data in several places on the disc, and use a head position sensitive method to determine which set of data should be accessed next.

The second line of defense is to distract the user from the slow seek times. Seek while the user is preoccupied with reading on-screen instructions or listening to Amiga music or sounds. Take advantage of the 96K of filesystem preloaded data. Have *something* happening on the screen to take the user's mind off the wait while the application waits for the new data to be read in from the CD disc.

The memory configuration of the standard CDTV is 1 Mbyte of Chip RAM. There is no Fast RAM on CDTV. This implies three things:

1. All accesses to RAM will face bus contention from the display. If you have a 640x200, 4 bitplane Hires screen, performance will decrease since your program is operating from Chip RAM where the contention is taking place.
2. Any requests by your application for Fast RAM will *fail*. There is almost never any need to specifically request Fast RAM on the standard Amiga, as the Exec memory allocation method will supply Fast RAM before Chip RAM if no memory type is otherwise specified. There is good reason never to request Fast RAM specifically, as this can cause an otherwise perfectly reasonable request to fail, as it does on the CDTV.
3. On a CDTV system, you have one large pool of memory. While this does give you a larger contiguous space (800K-900K), it also means that there is only one memory space to fragment. If you break it up into little pieces, it will be difficult to put back together until the next reboot. One of the goals of your application should be to keep the memory as unfragmented as possible. Fragmented memory can also hurt overall system performance because memory allocations can take longer.

The standard CDTV unit does not have an internal floppy drive, and while an external floppy drive can be connected to CDTV, you cannot count on one being there. This eliminates the usual Amiga method of storing data on a floppy. Instead, you will use the *bookmark.device*, which can save small amounts of application specific data in non-volatile RAM (NVR).

The CDTV NVR survives reboots, survives if the unit is turned off, but is erased if the unit is unplugged. It is important to remember that the amount of storage in a bookmark entry is very small compared to the storage space available on a floppy. Generally, you will need to adopt different storage techniques to fit your data into this space which will never be more than 1/16 the size of the total bookmark data area. For example, if the total size of the bookmark memory is 1K, then the largest entry allowed is 64 bytes. While your application could use multiple entries to get around this limitation, it is not generally recommended.

CDTV also has the capability for external "credit card" size RAM or ROM cards to be plugged into the front of the CDTV. These cards can be treated as additional storage for bookmarks, as RAM disks (or ROM disks) or even as additional system RAM. This is determined when the card is initialized. Most CDTV software will be distributed on CD discs, though it is possible to distribute special purpose software (like an extended audio panel) on one of these credit cards.

The infrared (IR) controller, which replaces the keyboard and mouse of the standard A500, must radically reshape the design of your software. The user can no longer enter a text string easily because the keypad on the standard controller has a limited number of keys. For movement, the controller has four cursor keys which can be used to emulate mouse movements or the action of a joystick. (There is a button on the controller to select joystick or mouse mode.) This means the normal Amiga method of selection by moving a pointer and selecting an object by clicking on the left mouse button is impractical because a pointer on a bitmap screen is very hard to control via

cursor keys to any degree of accuracy (or to any degree of speed). A few minutes playing with Workbench using the CDTV controller is generally enough to convince anyone of this. Instead, alternate selection methods must be used.

In the CDTV Preferences, for instance, there are a limited number of icons to select, and an equally limited number of screen positions for the pointer to be (in this case, the "pointer" is indicated by the flashing box surrounding the currently selected icon). To enter a text string, a videogame-like high score name entry method must be used. The base design of many Amiga programs depends on direct control of the cursor via the mouse and easy entry of text. On CDTV these basic assumptions must be reworked.

An important note on the use of the standard IR controller is that it has two modes—mouse emulation and joystick emulation. Currently, the keypad is only active in mouse mode. For best results, your application should work with the controller in *both* mouse and joystick modes. (In the *playerprefs.library*, there is an input handler to make this simple). Working with both modes avoids user frustration at pressing keys on an apparently dead controller and seemingly being ignored by the application.

Normally, when you eject a CD disc from CDTV, the system will reset. This is desirable in most cases; when the user ejects the CD, she is finished with that application and wishes to do something else. Resetting the system is the cleanest method of providing a clean slate for the next application because CDTV applications are self-booting and set up their own execution environment.

In certain special cases, however, this behavior is undesirable. An example is a program that allows the user to examine the Table of Contents of a disc. It would be impractical to make him reboot the program to look at another disc. To avoid the automatic reboot, your application will have to install a Changeint handler in the *cdtv.device* driver. Most applications, however, should allow the automatic reboot to occur.

This should probably go without saying, but on CDTV, avoid going directly to the hardware. Even though the characteristics of the current base machine are known, make sure your program is adaptable. Avoid software timing loops; if the processor speed is increased in future models, your application will break. Avoid hitting the CD disc controller directly; if the hardware were to change your application would stop working. Do not count on the current memory map; take advantage of additional RAM if it is available, and work with less RAM if some is unavailable. If the user has a SCSI drive hooked up, and is using 100K for buffers, please, try to continue to work. If the user has two floppies hooked up to an A690, there will be somewhat less RAM available for your application.

The standard CDTV unit is intended to be the beginning of a family. Each member of the family will have somewhat different characteristics. It is to your advantage and to our advantage if your application works on the entire family of products.

Software Differences Between CDTV and the A500

For the most part, CDTV has the same software as the A500 plus some extras. CDTV also has an additional 256K of CDTV system software, which contains the modules necessary to support the CDTV hardware. The current CDTV unit is equipped with Version 1.3 of the Amiga Operating System.

Prepare For The Future. With a probable operating system upgrade in CDTV's future, and the existence of the A690 CDTV add-on for the A500, CDTV software will need to operate under both 1.3 and 2.04. to operate under both 1.3 and 2.04.

The main differences between the CDTV software and the A500 software are as follows:

No GURUs

Possibly this, more than anything illustrates the difference between the expected audience for CDTV systems and for the A500. CDTV systems never GURU. A CDTV owner should never see the flashing red box after the failure of an application program or piece of system software. Instead, the CDTV will reset. The additional information that the alert message brings the A500 user was determined to be inappropriate and unnecessary to the CDTV user.

Watch For Multiple Resets. If you hear reports of your application resetting a CDTV many times, the application is involved in a GURU situation. Only the GURU has been removed. The *Software Error Task Held* requester is still present. Your application should turn off error requesters by setting the process `pr_WindowPtr` to -1 and handle them internally, instead.

Low Resolutions

The normal A500 is connected to an RGB monitor, and is set to a resolution of 640x200, and text is output in 80 columns. The normal CDTV is connected to the family television. 640x200 may not be appropriate, and 80 column text is never appropriate. Unlike an A500 where the user is right next to the monitor, CDTV will be used by a person sitting several feet or yards away from the screen. Your selection of resolutions must be appropriate to this situation. Rediscover the joys of Lores 320x200 screens, like the additional color and bandwidth. Make sure your text is readable on a television from several feet away.

ISO Filesystem

CD discs for CDTV are not in the standard AmigaDOS file format. CDTV supports an extended version of the ISO-9660 file format as its preferred format for CD discs. The characteristics of the ISO-9660 file format are somewhat different than that of the AmigaDOS file format. In fact, it is much like an extended MS-DOS format: It is very quick to get a directory; the time to find a particular file will vary depending on where a file is in the directory or on the disk so the access times between files in different directories can vary greatly; and the block allocation method used is well suited to loading large files quickly.

This is in direct contrast to the characteristics of the AmigaDOS filesystem. Many of the tricks used in Amiga applications to speed up access of files will actually slow down a CDTV application when accessing a CD disc. An application will probably have to be tuned for the best performance from a CD disc. The *ICOM* CDTV emulator card can be of great assistance when tuning, as it exactly emulates the characteristics of the CD drive, and is much quicker to use than burning gold test discs.

User Interface—Single Tasking, No Windows, One Screen

The user interface of a CDTV application should be different than that of an Amiga application. To the user, a CDTV system, unlike an A500 system, will typically seem to be doing only one thing at a time—running your application. Multitasking is still around, but it is in the background. The CDTV user will generally not be opening up multiple Shells, or running a

raytrace program in the background. There will usually be only one window present on the screen.

User controlled sliding screens will probably not be used for a while, if at all. This means that the standard Amiga intuition gadgets found in window and screen borders are not necessary nor desirable for use in a CDTV User Interface (UI). The IR controller does not lend itself to easy manipulation of those gadgets, so a different UI model is mandated. Even pulldown menus are affected by the CDTV UI choices, i.e., they are not to be used at all. To the user, a typical CDTV screen will be much less busy than a typical Amiga screen—lower resolution, fewer icons, less text, and fewer gadgets will give the impression of a simpler to use machine.

No Workbench

On a CDTV system, the average user will not encounter Workbench. Workbench is generally used to maintain files and launch applications. In general, there is little file maintenance involved on CD discs. Starting a CDTV application will normally occur from the Startup-Sequence on the application disc itself. In other words, CDTV applications should be designed as self-booting discs. The user will expect to insert the disc and have the application on that disc automatically start. In cases of discs containing multiple applications, you will need to provide a means of selecting the proper application to run, and the means to launch the application. Workbench should not be used for this purpose.

The startup-sequence of a CDTV application should be as short as possible. The user should not be left staring at a blank screen while a large application loads its data files; something must appear on the screen as quickly as possible.

CDTV Preferences

Unlike the A500, CDTV keeps some of its preferences settings, such as language and screen centering in the NVR. These settings are read by the *playerprefs.library*, and used automatically in the CDTV title screen and the screen saver. If your application wishes to take advantage of the same settings, it must access those settings itself, either directly through the *playerprefs.library*, or indirectly by placing the *BookIt* utility in the Startup-Sequence file for the application disk. The settings in *devs:System-Configuration* are used as initial settings, which are then modified by the CDTV Preferences stored in the NVR.

Playerprefs.library

The *playerprefs.library* provides some routines which can be useful in developing an application for CDTV. Some of the more important routines can also be accessed via external programs executed in the application disc Startup-Sequence. The *playerprefs.library* has routines for reading CDTV preferences, centering the screen, a screen saver, a joy/mouse controller handler, a keyclick handler, routines for color map manipulation, and for simple bitmap manipulation. In short, it contains the routines needed to operate the CDTV Preferences and Audio Panel programs. Several of the routines will be very useful to your application. The most useful can be invoked from the Startup-Sequence via the *BookIt* command. However, if your application calls the *playerprefs.library* directly, you can eliminate the need for the additional command in the startup-sequence.

CDTV Printer Preferences

The *prtprefs.library* provides the means for the user to make printer preferences settings such as the model of printer, paper size, and lines per page. The *prtprefs.library* includes a CDTV Printer Preferences editor with a similar look and feel to the CDTV Player Preferences program.

This library is not included in the CDTV system ROM, but is included on the CD disc of a CDTV application that wishes to provide printing capabilities to its users.

No keyboard

The CDTV IR Controller is much simpler than a standard Amiga keyboard, in that it has fewer keys. This requires restructuring applications so that the standard IR controller can be used *easily* to operate the application. Never count on the presence of an optional trackball, keyboard, or mouse. Most CDTV owners will have the base configuration. If the application is difficult to use without optional equipment, it may find its market limited.

Recommended CDTV Development Environments

One of the major advantages of CDTV development versus development for other multimedia platforms is the cost of the basic development platform. The Amiga offers all the tools and peripherals necessary for the majority of applications. This article describes the equipment necessary for CDTV developers, in function of their budget and specific requirements.

Minimum Configuration

For the developer with limited capital, here is the suggested minimum development configuration:

- Amiga 2000 (1 Mbyte chip RAM)
- A2058-2 memory expansion board
- A2091 SCSI disk controller
- 50 Mbyte SCSI disk
- A1084 monitor
- A520 (NTSC) or A521 (PAL) video adapter
- Inexpensive TV set
- CDTV player
- A1011 floppy disk drive

THE CPU

The Amiga 2000 has the same CPU (Motorola 68000) at the same clock speed as the CDTV. This assures equal performance between your development system and the target system, avoiding potential surprises when animations run slower on a 68000 than on the 68040 at 25Mhz that had been used for development and testing. It also has the same limited amount of RAM (1 Mbyte Chip RAM standard).

Two additional Mbytes of RAM will probably be necessary, as many of the development tools work much better with 3 Mbytes of RAM rather than just 1 Mbyte. A RAM disk is often useful to speed up compilations, for temporary storage, etc.

Memory expansion is available from numerous vendors. Commodore's A2058 memory board comes standard with 2 Mbytes, but can be expanded up to 8 Mbytes if necessary.

Kickstart 1.3... The first CDTV player was shipped with Kickstart 1.3 in ROM. Subsequent models of the CDTV family will ship with Kickstart 2.04 in ROM. The new version of the Kickstart provides numerous advantages to the CDTV developer. However, it also provides a disadvantage: applications must be able to support both versions of the operating system.

or 2.04? The Amiga 3000 and the A500 Plus have been shipped with KS 2.04. Systems like the A2500 and A3000 allow the developer to boot either under KS 1.3 or KS 2.04. Finally, KS 2.04 upgrade kits for CDTV players are available to CDTV developers for testing their applications. It is vital to select one of these options and to test your application under both versions of the operating system prior to mastering CDs.

SCSI CONTROLLERS AND DISK DRIVES

A SCSI disk controller is obligatory. Commodore's A2091 board can be used as a hard card—a 3.5" SCSI disk can be mounted directly on the board. It also has an external 25-pin SCSI connector for connecting external drives. This feature often comes in handy as you expand your development environment, adding new SCSI disk drives, tape backup units, etc.

If you use the A2091, make sure you have the latest revision of the ROMs on the controller. As of this writing, the most recent revision is 6.6.

The size of the SCSI disk attached to the controller should correspond to the application at hand. A game may only require 2 Mbytes of data, while an atlas may need 450 Mbytes.

The following drives have been tested and approved:

Quantum 50, 105, 200 Mbyte
Seagate 410 Mbyte (ref. ST2502N)
Seagate 600 Mbyte (ref. ST4766N)

MONITORS AND TV SETS

Commodore's A1084 monitor is recommended as an inexpensive solution for development work. It can accept both RGB and composite video signals, and it includes stereo speakers. The 1084 exists in both NTSC and PAL versions.

A standard TV set is strongly recommended, the cheaper the better. The target CDTV player will nearly always be connected to a TV set, not to an RGB monitor. When an RGB signal is encoded for RF or composite output, colors are saturated, shadows appear and text becomes much more difficult to read. Testing on a TV set during product storyboarding and prototyping can save enormous amounts of time later.

Why a cheap TV? If your product looks good on an inexpensive TV set, it will look *wonderful* your big-screen, surround sound set. Many of the TV sets in use are 5 to 10 years of age. Many of them are poorly adjusted for colors, image centering, contrast, etc. Using an inexpensive set for development is a constant reminder of the reality of many consumer entertainment centers.

The least expensive way to connect a TV set to your Amiga is via the A520 (NTSC) or A521 (PAL) encoder. This small module plugs into the RGB port of your Amiga, and generates both RF and composite video signals. You can connect the RF signal to your TV set, or the composite signal may be connected to the A1084 monitor.

CDTV PLAYER AND EXTERNAL FLOPPY

A CDTV player is required for testing your product. The standard player is sufficient for a minimum configuration. However, it is recommended to include extra accessories when possible. Your code should accept input smoothly from a trackball as well as from the remote control. Video applications should be tested using a CDTV genlock.

Test On The Tube. Make sure to connect your player to your TV set for testing, not to your RGB monitor.

The external A1011 floppy disk drive is vital for product testing. With this drive attached to your CDTV player, you can boot your application from a floppy disk and test various parts of your code, such as the routines to handle input from the remote, or to synchronize CD audio tracks. When you have produced a test disc, you can use the data on the test disc (which usually does not change) and uncover the bugs lurking in your code (which usually does).

Mid-sized Configuration

The following configuration is recommended for the developer with an average budget:

- Amiga 2500 (5 Mbytes RAM total)
- A2091 SCSI disk controller
- 50 Mbyte SCSI disk
- 450 Mbyte 2nd SCSI disk
- A2320 Video Display Enhancer
- A1950 Multiscan monitor
- A3070 streamer tape drive
- A520 (NTSC) or A521 (PAL) video adapter
- Inexpensive TV set
- CDTV player
- A1011 floppy disk drive

THE CPU

The Amiga 2500 contains two CPUs: the Motorola 68000 at 7 Mhz, and a 68030 at 25 Mhz. Developing in 68030 mode is much more comfortable than on a 68000, yet it is possible to revert to the 68000 for testing your application. Simply press both mouse buttons while booting, then select 68000 mode from the menu. Furthermore, in 68000 mode only 1 Mbyte of RAM is available to the system (just like on the CDTV player).

SCSI CONTROLLERS AND DISK DRIVES

When more disk capacity is required, at least two SCSI disk drives are recommended. One disk can be used for the operating system, tools, compilers, etc. The other disk may contain data, source code, etc. If you use a large hard disk, consider creating two or more partitions on the drive. This provides added security in case of a read/write error (only one partition needs to be re-formatted). Certain development tools (the CD-XL toolkit, audio capture boards, etc.) do intensive reads and writes to the disk, increasing the likelihood of disk errors.

Make sure to verify the SCSI device number on each drive you connect to the system. If two devices have the same number, the system will not boot. Device numbers are usually determined by jumpers or DIP switches on the SCSI drives.

If you purchase an external SCSI disk drive, consider purchasing an external housing as well. The housing (delivered standard with some drives) contains the power supply and external connectors to facilitate daisy-chaining the drives.

BACKUP SYSTEMS

If you opt for a large disk, you should seriously consider purchasing some sort of backup system. Floppy disks may suffice for a 20 Mbyte hard drive; a 50 Mbyte drive requires 57 floppies; a 450 Mbyte drive requires a tape backup.

Another advantage of tape backup systems appears late in the development cycle, during pre-mastering. When you are ready to cut a test CD-ROM, you must send your data to a pre-mastering center with write-once equipment. Many developers have sent their SCSI hard disk drives to the pre-mastering facility. If you have a tape system, you may send along a tape, and continue developing while your product is pre-mastered.

The A3070 tape drive from Commodore provides data protection at a reasonable cost. Backup is reasonably fast (7 Mbytes/minute). Third parties have developed utility software to back-up and restore to this (and other) SCSI tape units. *AmiBack*, from Moonlighter Software, has been tested successfully.

Some developers have used removable SCSI drives successfully. Syquest drives are an inexpensive way to back up a hard disk drive. However, caution is advised, as problems have been reported from some developers using Syquest drives.

MONITORS AND TV SETS

The A1950 multiscan monitor, combined with the A2320 display enhancer, provide a stable, non-flicker image in interlaced mode. However, as discussed previously, prototyping, screen design and product testing should be performed on a TV set, not on an A1950.

High-end configuration

If sufficient funding is available, the following configuration will provide maximum comfort and productivity for the developer:

- Amiga 2500 or Amiga 3000 with 9 Mbytes RAM
- A2091 SCSI disk controller (A2500 only)
- 50 Mbyte SCSI disk
- 650 Mbyte SCSI disk drive
- A3070 or DAT tape drive
- A1950 monitor
- A520 (NTSC) or A521 (PAL) video adapter
- Inexpensive TV set
- CDTV player
- A1011 floppy disk drive
- CTrac emulator

THE CPU

The Amiga 2500 remains the development system of choice, since its two processors provide both a fast development processor and a slow target emulation system. However, the A3000 may also be used for development. The A3000 includes many of the optional features of the A2500 (SCSI controller, A2630 Video Display Adapter, memory expansion capability) as standard items on the mother board. Thus a fully-configured A3000 is usually less expensive than an equivalent A2500.

If you choose to develop on an A3000, you should take special precautions to avoid 68030-specific code, and to test your application on a system at 7 Mhz.

9 Mbytes of RAM are useful for grabbing large images, and for processing 24-bit graphics.

SCSI CONTROLLERS AND DISK DRIVES

The A3000 includes a SCSI controller directly on the motherboard, as well as an external 25-pin SCSI connector. However, it has no room for an internal 5.25" device.

DAT DRIVES

DAT drives provide numerous advantages over other tape backup systems:

- They can be used to record and play back CD-quality audio, at 44.1 Khz
- Their capacity (1.2 Gbytes) is appreciable
- They are commonly used by mastering facilities

CTRAC EMULATOR

The *CTrac* emulator, from ICOM Simulations, is an extremely useful tool for CDTV development. This system lets you create an ISO 9660 CD-ROM image on a SCSI hard disk, including error correction, sub-code information, CD-DA and CD+G tracks. You then connect your CDTV player to the emulation board—instead of to the CD-ROM mechanism in the player—and run your application directly off the SCSI disk image.

This is a full emulation—the application actually runs in the memory of the CDTV player, *not* the Amiga host. It fully emulates CD-ROM seek times and data transfer rates.

This board saves time (and money) for developers by reducing the number of test discs which must be produced. You can immediately test response times, seek times, and system throughput. The ability to play back CD audio tracks lets you precisely emulate mixed-mode discs.

Developer's Introduction

This is the general introduction to the Amiga family of computers. It is included here for developers new to the Amiga platform. For the current version of CDTV, only the portions of this introduction dealing with the 68000 are relevant.

The Amiga family of computers consists of several models, each of which has been designed on the same premise—to provide the user with a low-cost computer that features high-cost performance. The Amiga does this through the use of custom silicon hardware that yields advanced graphics and sound features.

There are four basic models that make up the Amiga computer family: the A500, A1000, A2000, and A3000. Though the models differ in price and features, they have a common hardware nucleus that makes them software compatible with one another. This chapter describes the Amiga's hardware components and gives a brief overview of its graphics and sound features.

Components of the Amiga

These are the hardware components of the Amiga:

- Motorola MC68000 16/32-bit main processor. The Amiga also supports the 68010, 68020, and 68030 processors as an option. The A1000, A500 and A2000 contain the 68000, while the A3000 utilizes the 68030 processor.
- Custom graphics and audio chips with DMA capability. All Amiga models are equipped with three custom chips named Paula, Agnus, and Denise which provide for superior color graphics, digital audio, and high-performance interrupt and I/O handling. The custom chips can access up to 2MB of memory directly without using the 68000 CPU.
- From 256K to 2 MB of RAM expandable to a total of 8 MB (over a gigabyte on the Amiga 3000).
- 512K of system ROM containing a real time, multitasking operating system with sound, graphics, and animation support routines. (V1.3 and earlier versions of the OS used 256K of system ROM.)
- Built-in 3.5 inch double sided disk drive with expansion floppy disk ports for connecting up to three additional disk drives (either 3.5 inch or 5.25 inch, double sided).
- SCSI disk port for connecting additional SCSI disk drives (A3000 Only).
- Fully programmable parallel and RS-232-C serial ports.
- Two button opto-mechanical mouse and two reconfigurable controller ports (for mice, joysticks, light pens, paddles, or custom controllers).

- A professional keyboard with numeric keypad, 10 function keys, and cursor keys. A variety of international keyboards are also supported.
- Ports for analog or digital RGB output (all models), monochrome video (A500 and A2000), composite video (A1000), and VGA-style multiscan video (A3000).
- Ports for left and right stereo audio from four special purpose audio channels.
- Expansion options that allow you to add RAM, additional disk drives (floppy or hard), peripherals, or coprocessors.

THE MC68000 AND THE AMIGA CUSTOM CHIPS

The Motorola MC68000 microprocessor is the CPU used in the A1000, the A500, and the A2000. The 68000 is a 16/32-bit microprocessor; internal registers are 32 bits wide, while the data bus and ALU are 16 bits. The 68000's system clock speed is 7.15909 MHz on NTSC systems (USA) or 7.09379 MHz on PAL systems (Europe). These speeds can vary when using an external system clock, such as from a genlock board.

The 68000 has an address space of 16 megabytes. In the Amiga, the 68000 can address up to 9 megabytes of random access memory (RAM).

In the A3000, the Motorola MC68030 microprocessor is the CPU. This is a full 32-bit microprocessor with a system clock speed of 16 or 25 megahertz. The 68030 has an address space of 4 gigabytes. In the A3000, over a gigabyte of RAM can be addressed.

In addition to the 680x0, all Amiga models contain special purpose hardware known as the *custom chips* that greatly enhance system performance. The term *custom chips* refers to the three integrated circuits which were designed specifically for the Amiga computer. These three custom chips, named Paula, Agnus, and Denise, each contain the logic to handle a specific set of tasks such as video, audio, or I/O.

Because the custom chips have DMA capability, they can access memory without using the 680x0 CPU—this frees the CPU for other types of operations. The division of labor between the custom chips and the 680x0 gives the Amiga its power; on most other systems the CPU has to do everything.

The memory shared between the Amiga's CPU and the custom chips is called *Chip memory*. The more Chip memory the Amiga has, the more graphics, audio, and I/O data it can operate on without the CPU being involved. All Amigas can access at least 512K of Chip memory.

The latest version of the custom chips, known as the *Enhanced Chip Set* or *ECS*, can handle up to 2 MB of memory and has other advanced features.

Although there are different versions of the Amiga's custom chips, all versions have some common features. Among other functions, the custom chips provide the following:

- Bitplane generated, high resolution graphics capable of supporting both PAL and NTSC video standards.

NTSC systems. On NTSC systems, the Amiga typically produces a 320 by 200 non-interlaced or 320 by 400 interlaced display in 32 colors. A high resolution mode provides a 640 by 200 non-interlaced or 640 by 400 interlaced display in 16 colors.

PAL systems. On PAL systems, the Amiga typically produces a 320 by 256 non-interlaced or 320 by 512 interlaced display in 32 colors. High resolution mode provides a 640 by 256 non-interlaced or 640 by 512 interlaced display in 16 colors.

The design of the Amiga's display system is very flexible and there are many other modes available. Hold-and-modify (HAM) mode allows for the display of up to 4,096 colors on screen simultaneously. Overscan mode allows the creation of higher resolution displays specially suited for video and film applications. Displays of arbitrary size, larger than the visible viewing area can be created. Amigas which contain the Enhanced Chip Set (ECS) support Productivity mode giving displays of 640 by 480, non-interlaced with 4 colors from a palette of 64.

- A custom graphics coprocessor, called *the Copper*, that allows changes to most of the special purpose registers in synchronization with the position of the video beam. This allows such special effects as mid-screen changes to the color palette, splitting the screen into multiple horizontal slices each having different video resolutions and color depths, beam-synchronized interrupt generation for the 680x0, and more. The coprocessor can trigger many times per screen, in the middle of lines, and at the beginning or during the blanking interval. The coprocessor itself can directly affect most of the registers in the other custom chips, freeing the 680x0 for general computing tasks.
- 32 system color registers, each of which contains a 12-bit number as four bits of red, four bits of green, and four bits of blue intensity information. This allows a system color palette of 4,096 different choices of color for each register.
- Eight reusable 16-bit wide sprites with up to 15 color choices per sprite pixel (when sprites are paired). A sprite is an easily movable graphics object whose display is entirely independent of the background (called a playfield); sprites can be displayed over or under this background. A sprite is 16 low resolution pixels wide and an arbitrary number of lines tall. After producing the last line of a sprite on the screen, a sprite DMA channel may be used to produce yet another sprite image elsewhere on screen (with at least one horizontal line between each reuse of a sprite processor). Thus, many small sprites can be produced by simply reusing the sprite processors appropriately.
- Dynamically controllable inter-object priority, with collision detection. This means that the system can dynamically control the video priority between the sprite objects and the bitplane backgrounds (playfields). You can control which object or objects appear over or under the background at any time. Additionally, you can use system hardware to detect collisions between objects and have your program react to such collisions.
- Custom bit blitter used for high speed data movement, adaptable to bitplane animation. The blitter has been designed to efficiently retrieve data from up to three sources, combine the data in one of 256 different possible ways, and optionally store the combined data in a destination area. The bit blitter, in a special mode, draws patterned lines into rectangularly organized memory regions at a speed of about 1 million dots per second; and it can efficiently handle area fill.
- Audio consisting of four digital channels with independently programmable volume and sampling rate. The audio channels retrieve their control and sample data via DMA. Once started, each channel can automatically play a specified waveform without further processor interaction. Two channels are directed into each of the two stereo audio outputs. The audio channels may

be linked together to provide amplitude or frequency modulation or both forms of modulation simultaneously.

- DMA controlled floppy disk read and write on a full track basis. This means that the built-in disk can read over 5600 bytes of data in a single disk revolution (11 sectors of 512 bytes each).

AMIGA MEMORY SYSTEM

As mentioned previously, the custom chips have DMA access to RAM which allows them to perform graphics, audio, and I/O chores independently of the CPU. This shared memory that both the custom chips and the CPU can access directly is called *Chip memory*.

The custom chips and the 680x0 CPU share Chip memory on a fully interleaved basis. Since the 680x0 only needs to access the Chip memory bus during each alternate clock cycle in order to run full speed, the rest of the time the Chip memory bus is free for other activities. The custom chips use the memory bus during these free cycles, effectively allowing the CPU to run at full speed most of the time.

There are some occasions though when the custom chips steal memory cycles from the 680x0. In the higher resolution video modes, some or all of the cycles normally used for processor access are needed by the custom chips for video refresh. In that case, the Copper and the blitter in the custom chips steal time from the 680x0 for jobs they can do better than the 680x0. Thus, the system DMA channels are designed with maximum performance in mind.

Even when such cycle stealing occurs, it only blocks the 680x0's access to the internal, shared memory. The custom chips cannot steal cycles when the 680x0 is using ROM or external memory, also known as *Fast memory*.

The DMA capabilities of the custom chips vary depending on the version of the chips and the Amiga model. The original custom chip set found in the A1000 could access the first 512K of RAM. Most A1000s have only 512K of RAM so some of the Chip RAM is used up for operating system overhead.

A later version of the custom chips found in early A500s and A2000s replaced the original Agnus chip (8361) with a newer version called *Fat Agnus* (8370/8371). The Fat Agnus chip has DMA access to 512K of Chip memory, just like the original Agnus, but also allows an additional 512K of internal *slow memory* or *pseudo-fast memory* located at (\$00C0 0000). Since the slow memory can be used for operating system overhead, this allows all 512K of Chip memory to be used by the custom chips.

The name *slow memory* comes from the fact that bus contention with the custom chips can still occur even though only the CPU can access the memory. Since slow memory is arbitrated by the same gate that controls Chip memory, the custom chips can block processor access to slow memory in the higher resolution video modes.

The latest version of Agnus and the custom chips found in most A500s and A2000s is known as the *Enhanced Chip Set* or ECS. ECS Fat Agnus (8372A) can access up to one megabyte of Chip memory. It is pin compatible with the original Fat Agnus (8370/8371) found in earlier A500 and A2000 models. In addition, ECS Fat Agnus supports both the NTSC and PAL video standards on a single chip.

In the A3000, the Enhanced Chip Set can access up to two megabytes of Chip memory.

The amount of Chip memory is important since it determines how much graphics, audio, and disk data the custom chips can operate on without the 680x0 CPU. The table below summarizes the basic memory configurations of the Amiga.

	Chip RAM (base model)	Maximum Chip RAM	Total RAM (base model)	Maximum Total RAM
Amiga 1000	256K	512K	256K	9 MB
Amiga 500	512K	1 MB	1 MB	9 MB
Amiga 2000	512K	1 MB	1 MB	9 MB
Amiga 3000	1 MB	2 MB	2 MB	over 1 GB

Another primary feature of the Amiga hardware is the ability to dynamically control which part of the Chip memory is used for the background display, audio, and sprites. The Amiga is not limited to a small, specific area of RAM for a frame buffer. Instead, the system allows display bitplanes, sprite processor control lists, coprocessor instruction lists, or audio channel control lists to be located anywhere within Chip memory.

This same region of memory can be accessed by the bit blitter. This means, for example, that the user can store partial images at scattered areas of Chip memory and use these images for animation effects by rapidly replacing on screen material while saving and restoring background images. In fact, the Amiga includes firmware support for display definition and control as well as support for animated objects embedded within playfields.

Floppy disk storage is provided by a built-in, 3.5 inch floppy disk drive. Disks are 80 track, double sided, and formatted as 11 sectors per track, 512 bytes per sector (over 900,000 bytes per disk). The disk controller can read and write 320/360K IBM PCTM (MS-DOSTM) formatted 3.5 or 5.25 inch disks, and 640/720K IBM PC (MS-DOS) formatted 3.5 inch disks.

Up to three extra 3.5 inch or 5.25 inch disk drives can be added to the Amiga. The A2000 and A3000 also provide room to mount floppy or hard disks internally. The A3000 has a built-in hard disk drive and an on-board SCSI controller which can handle two internal drives and up to seven external SCSI devices.

The Amiga has a full complement of dedicated I/O connectors. The circuitry for some of these peripherals resides on the Paula custom chip while the Amiga's two 8520 CIA chips handle other I/O chores not specifically assigned to any of the custom chips. These include modem control, disk status sensing, disk motor and stepping control, ROM enable, parallel input/output interface, and keyboard interface.

The Amiga includes a standard RS-232-C serial port for external serial input/output devices such as a modem, MIDI interface, or printer. A programmable, Centronics-compatible parallel port supports parallel printers, audio digitizers, and other peripherals.

The Amiga also includes a two-button, opto-mechanical mouse plus a keyboard with numeric keypad, cursor controls, and 10 function keys in the base system. A variety of international keyboards are supported. Many other input options are available. Other types of controllers can be attached through the two controller ports on the base unit including joysticks, keypads, trackballs, light pens, and graphics tablets.

System Expandability And Adaptability

New peripheral devices may be easily added to all Amiga models. These devices are automatically recognized and used by system software through a well defined, well documented linking procedure called *AUTOCONFIG*[™]. *AUTOCONFIG* is short for automatic configuration and is the process that allows memory or I/O space for an expansion board to be dynamically allocated by the system at boot time. Unlike some other systems, there is no need to set DIP switches to select an address space from a fixed range reserved for expansion devices.

On the A500 and A1000 models, peripheral devices can be added using the Amiga's 86-pin expansion connector. Peripherals include hard disk controllers and drives, and additional external RAM. Extra floppy disk units may be added from a connector at the rear of the unit.

The A2000 and A3000 models provide the user with the same features as the A500 or A1000, but with the added convenience of simple and extensive expandability through the Amiga's 100-pin Zorro expansion bus.

The A2000 contains 7 internal slots and the A3000 contains 4 internal slots plus a SCSI disk controller that allow many types of expansion devices to be quickly and easily added inside the machine. Available options include RAM boards, coprocessors, hard disk controllers, video cards, and I/O ports.

The A2000 and A3000 also support the special Bridgeboard[™] coprocessor card. This provides a complete IBM PC[™] on a card and allows the Amiga to run MS-DOS[™] compatible software, while simultaneously running native Amiga software. In addition, both machines have expansion slots capable of supporting standard, IBM PC[™] style boards.

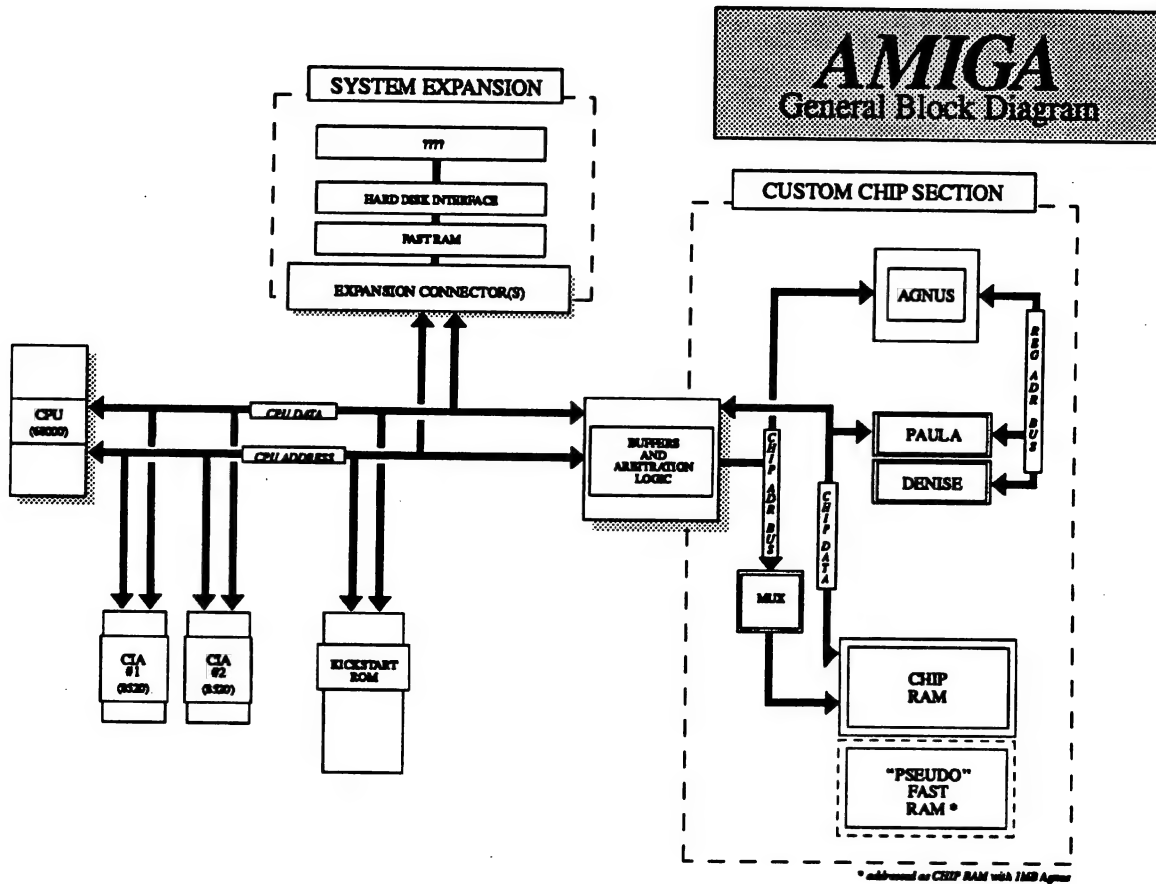
VCR And Direct camera Interface

In addition to the connectors for monochrome composite, and analog or digital RGB monitors, the Amiga can be expanded to include a VCR or camera interface. With a genlock board, the system is capable of synchronizing with an external video source and replacing the system background color with the external image. This allows development of fully integrated video images with computer generated graphics. Laser disk input is accepted in the same manner.

The A2000 and A3000 models also provide a special internal slot designed for video applications. This allows the Amiga to use low-cost video expansion boards such as genlocks and frame-grabbers.

Amiga System Block Diagram

The diagram below highlights the major hardware components of the Amiga's architecture. Notice that there are two separate buses, one that only the CPU can access (Fast memory) and another one that can the custom chips share with the CPU (Chip memory).





General Amiga Development Guidelines

The environment of the Amiga computer is quite different than that of many other systems. The Amiga is a multitasking platform, which means multiple programs can run on a single machine simultaneously. However, for multitasking to work correctly, care must be taken to ensure that programs do not interfere with one another. It also means that certain guidelines must be followed during programming.

- Check for memory loss. Operate your program, then exit. Write down the amount of free memory. Repeat the operation of your program and exit. The amount of free memory remaining should be exactly the same. Any difference indicates a serious problem in your cleanup. (Beware when checking the amount of free memory - some memory loss is normal the first time you open a device or disk-based library because the system has to allocate memory to accommodate them. That is why you should run the program once before checking free memory.) The tool *Drip* is useful in tracking this; the programs *Snoop* and *SnoopStrip* can be used to determine which allocation is not getting freed.
- Use all of the program debugging and stress tools that are available when writing and testing your code. New debugging tools such as *Enforcer*, *MungWall* and *Scratch* can help find uninitialized pointers, attempted use of freed memory and misuse of scratch registers or condition codes (even in programs that appear to work perfectly).
- Always make sure you actually get any system resource that you ask for. This applies to memory, windows, screens, file handles, libraries, devices, ports, etc. Where an error value or return is possible, ensure that there is a reasonable failure path. Many poorly written programs will appear to be reliable, until some error condition (such as memory full or a disk problem) causes the program to continue with an invalid or null pointer, or branch to untested error handling code.
- Always clean up after yourself. This applies for both normal program exit and program termination due to error conditions. Anything that was opened must be closed, anything allocated must be deallocated. It is generally correct to do closes and deallocations in reverse order of the opens and allocations. Be sure to check your development language manual and startup code; some items may be closed or deallocated automatically for you, especially in abort conditions. If you write in the C language, make sure your code handles Ctrl-C properly.
- Remember that memory, peripheral configurations, and ROMs differ between models and between individual systems. Do not make assumptions about memory address ranges, storage device names, or the locations of system structures or code. Never call ROM routines directly. Beware of any example code you find that calls routines at addresses in the \$F0 0000-\$FF FFFF range. These are ROM routines and they will move with every

OS release. The only supported interface to system ROM code is through the library, device, and resource calls.

- Never assume library bases or structures will exist at any particular memory location. The only absolute address in the system is \$0000 0004, which contains a pointer to the *exec.library* base. Do not modify or depend on the format of private system structures. This includes the poking of copper lists, memory lists, and library bases.
- Never assume that programs can access hardware resources directly. Most hardware is controlled by system software that will not respond well to interference from other programs. Shared hardware requires programs to use the proper sharing protocols. Use the defined interface; it is the best way to ensure that your software will continue to operate on future models of the Amiga.
- Never access shared data structures directly without the proper mutual exclusion (locking). Remember that other tasks may be accessing the same structures.
- The system does not monitor the size of a program's stack. Take care that your program does not cause stack overflow, and provide enough extra stack space for the possibility that future revisions of system functions might require additional stack space.
- Never use a polling loop to test signal bits. If your program waits for external events like menu selection or keystrokes, do not bog down the multitasking system by busy-waiting in a loop. Instead, let your task go to sleep by `Wait()`ing on its signal bits. For example:

```
signals = (ULONG)Wait((1<<windowPtr->UserPort->mp_SigBit) |
                    (1<<consoleMsgPortPtr->mp_SigBit));
```

This turns the signal bit number for each port into a mask, then combines them as the argument for the *exec.library/Wait()* function. When your task wakes up, handle all of the messages at each port where the `SigBit` is set. There may be more than one message per port, or no messages at the port. Make sure that you `ReplyMsg()` to all messages that are not replies themselves. If you have no signal bits to `Wait()` on, use `Delay()` or `WaitTOF()` to provide a measured delay.

- Tasks (and processes) execute in 680x0 user mode. Supervisor mode is reserved for interrupts, traps, and task dispatching. Take extreme care if your code executes in supervisor mode. Exceptions while in supervisor mode are deadly.
- Most system functions require a particular execution environment. All DOS functions and any functions that might call DOS (such as the opening of a disk-resident library, font, or device) can only be executed from a process. A task is not sufficient. Most other ROM kernel functions may be executed from tasks. Only a few may be executed from interrupts.
- Never disable interrupts or multitasking for long periods. If you use `Forbid()` or `Disable()`, you should be aware that execution of any system function that performs the `Wait()` function will temporarily suspend the `Forbid()` or `Disable()` state, and allow multitasking and interrupts to occur. Such functions include almost all forms of DOS and device I/O, including common "stdio" functions like `printf()`.
- Never tie up system resources unless it is absolutely necessary. For example, if your program does not require constant use of the printer, open the *printer.device* only when you need it. This will allow other tasks to use the printer while your program is running. You must provide a reasonable error response if a resource is not available when you need it.

- All data for the custom chips must reside in Chip memory (type MEMF_CHIP). This includes bitplanes, sound samples, trackdisk buffers, and images for sprites, bobs, pointers, and gadgets. The AllocMem() call takes a flag for specifying the type of memory. A program that specifies the wrong type of memory may appear to run correctly because many Amigas have only Chip memory. (On all models of the Amiga, the first 512K of memory is Chip memory and in some later models, Chip memory may occupy the first one or two megabytes).

However, once expansion memory has been added to an Amiga (type MEMF_FAST), any memory allocations will be made in the expansion memory area by default. Hence, a program can run correctly on an unexpanded Amiga which has only Chip memory while crashing on an Amiga which has expanded memory. A developer with only Chip memory may fail to notice that memory was incorrectly specified.

Most compilers have options to mark specific data structures or object modules so that they will load into Chip RAM. Some older compilers provide the Atom utility for marking object modules. If this method is unacceptable, use the AllocMem() call to dynamically allocate Chip memory, and copy your data there.

When making allocations that do not require Chip memory, do not explicitly ask for Fast memory. Instead ask for memory type MEMF_PUBLIC or 0L as appropriate. If Fast memory is available, you will get it.

- Never use software delay loops! Under the multitasking operating system, the time spent in a loop can be better used by other tasks. Even ignoring the effect it has on multitasking, timing loops are inaccurate and will wait different amounts of time depending on the specific model of Amiga computer. The *timer.device* provides precision timing for use under the multitasking system and it works the same on all models of the Amiga. The AmigaDOS Delay() function or the *graphics.library*/WaitTOF() function provide a simple interface for longer delays. The 8520 I/O chips provide timers for developers who are bypassing the operating system (see the Amiga Hardware Reference Manual for more information).
- Always obey structure conventions!

All non-byte fields must be word-aligned. Longwords should be longword-aligned for performance.

All address pointers should be 32 bits (not 24 bits). The upper byte must never be used for data.

Fields that are not defined to contain particular initial values must be initialized to zero. This includes pointer fields.

All reserved or unused fields must be initialized to zero for future compatibility.

Data structures to be accessed by the custom chips, public data structures (such as a task control block), and structures which must be longword aligned must NOT be allocated on a program's stack.

Dynamic allocation of structures with AllocMem() provides longword aligned memory of a specified type with optional initialization to zero, which is useful in the allocation of structures.

For 68010/68020/68030/68040 compatibility

Special care must be taken to be compatible with the entire family of 68000 processors:

- Do not use the upper 8 bits of a pointer for storing unrelated information. The 68020, 68030, and 68040 use all 32 bits for addressing.
- Do not use signed variables or signed math for addresses.
- Do not use software delay loops, and do not make assumptions about the order in which asynchronous tasks will finish.
- The stack frame used for exceptions is different on each member of the 68000 family. The type identification in the frame must be checked! In addition, the interrupt autovectors may reside in a different location on processors with a VBR register.
- Do not use the MOVE SR,<dest> instruction! This 68000 instruction acts differently on other members of the 68000 family. If you want to get a copy of the processor condition codes, use the *exec.library/GetCC()* function.
- Do not use the CLR instruction on a hardware register which is triggered by Write access. The 68020 CLR instruction does a single Write access. The 68000 CLR instruction does a Read access first, then a Write access. This can cause a hardware register to be triggered twice. Use MOVE.x #0, <address> instead.
- Self-modifying code is strongly discouraged. All 68000 family processors have a pre-fetch feature. This means the CPU loads instructions ahead of the current program counter. Hence, if your code modifies or decrypts itself just ahead of the program counter, the pre-fetched instructions may not match the modified instructions. The more advanced processors prefetch more words. If self-modifying code must be used, flushing the cache is the safest way to prevent troubles.
- The 68020, 68030 and 68040 processors all have instruction caches. These caches store recently used instructions, but do not monitor writes. After modifying or directly loading instructions, the cache must be flushed. See the *exec.library/CacheClearU()* Autodoc for more details. If your code takes over the machine, flushing the cache will be trickier. You can account for the current processors, and hope the same techniques will work in the future:

```

CACRF_ClearI    EQU      $0008    ;Bit for clear instruction cache
;
;Supervisor mode only. Use only if you have taken
;over the machine. Read and store the ExecBase
;processor AttnFlags flags at boot time, call this
;code only if the "68020 or better" bit was set.
;
ClearICache:    dc.w      $4E7A,$0002    ;MOVEC CACR,D0
                tst.w      d0           ;movec does not affect CC
                bmi.s      cic_040      ;A 68040 with enabled cache!
                ori.w      #CACRF_ClearI,d0
                dc.w      $4E7B,$0002    ;MOVEC D0,CACR
                bra.s      cic_exit
cic_040:        dc.w      $f458          ;CPUSHA (IC)
cic_exit:

```


Hardware Programming Guidelines

If you find it necessary to program the hardware directly, then it is your responsibility to write code that will work correctly on the various models and configurations of the Amiga. Be sure to properly request and gain control of the hardware resources you are manipulating, and be especially careful in the following areas:

- Kickstart 2.0 uses the 8520 Complex Interface Adaptor (CIA) chips differently than 1.3 did. To ensure compatibility, you must always ask for CIA access using the *cia.resource/AddICRVector()* and *RemICRVector()* functions. Do not make assumptions about what the system might be using the CIA chips for. If you write directly to the CIA chip registers, do not expect system services such as the *trackdisk.device* to function. If you are leaving the system up, do not read or write to the CIA Interrupt Control Registers directly; use the *cia.resource/AbleICR()*, and *SetICR()* functions. Even if you are taking over the machine, do not assume the initial contents of any of the CIA registers or the state of any enabled interrupts.
- All custom chip registers are Read-only or Write-only. Do not read Write-only registers, and do not write to Read-only registers.
- Never write data to, or interpret data from the unused bits or addresses in the custom chip space. To be software-compatible with future chip revisions, all undefined bits must be set to zeros on writes, and must be masked out on reads before interpreting the contents of the register.
- Never write past the current end of custom chip space. Custom chips may be extended or enhanced to provide additional registers, or to use bits that are currently undefined in existing registers.
- Never read, write, or use any currently undefined address ranges or registers. The current and future usage of such areas is reserved by Commodore and is subject to change.
- Never assume that a hardware register will be initialized to any particular value. Different versions of the OS may leave registers set to different values. Check the Amiga Hardware Reference Manual to ensure that you are setting up all the registers that affect your code.

Additional Assembler Development Guidelines

If you are writing in assembly language there are some extra rules to keep in mind in addition to those listed above.

- Never use the TAS instruction on the Amiga. System DMA can conflict with this instruction's special indivisible read-modify-write cycle.
- System functions must be called with register A6 containing the library or device base. Libraries and devices assume A6 is valid at the time of any function call. Even if a particular function does not currently require its base register, you must provide it for compatibility with future system software releases.
- Except as noted, system library functions use registers D0, D1, A0, and A1 as scratch registers and you must consider their former contents to be lost after a system library call. The contents of all other registers will be preserved. System functions that provide a result will return the result in D0.

- Never depend on processor condition codes after a system call. The caller must test the returned value before acting on a condition code. This is usually done with a TST or MOVE instruction.

2.0 Compatibility Problem Areas

General Compatibility Problem Areas

One sure fire way to write incompatible software is to fail to follow the Amiga programming guidelines listed in the beginning of your *Amiga ROM Kernel* and *Amiga Hardware* manuals. Please read the guidelines and follow them!

The following improper Amiga programming practices are likely to fail on new ROMs or hardware:

- Calling ROM code directly.
- Directly or indirectly reading or writing random memory addresses or low memory (especially location zero) due to improperly initialized pointers or structures. Use *Mungwall* and *Enforcer* when writing and testing your code!
- Assuming addresses/location/amounts of RAM or system structures.
- Requiring all free RAM.
- Mishandling 32-bit addresses. For example, using signed math or signed comparisons on addresses, or doing a BOOL or WORD test to determine if a pointer is non-zero.
- Overwriting memory allocations. With 32-bit addresses, a 1-byte overwrite of a string array can wipe out the high byte of a pointer or stack return address. This bug could go unnoticed on a 24-bit address machine (e.g., A500, A2500, etc.) but crash the system or cause other problems on an A3000.
- Shaving stack size too close. System function stack usage changes with each OS release.
- Improper flags or garbage in system structures. A bit that means nothing under one OS may drastically change the behavior of a function in a newer version of the OS. Clear structures before using, and use correct flags.
- Passing garbage in previously unused upper bytes of function arguments (for example, the upper word of the ULONG AvailFonts() Flags parameter).
- Improper register or condition code handling. Do not assume registers D0-D1/A0-A1 are preserved after system calls! Some function calls happen to preserve some registers. This can change in any revisions of the OS. In some cases we have modified the values returned in registers to keep certain applications from failing under 2.0. We do not guarantee those modifications will remain in place. Condition codes are also in an undefined state on the return from a system call. Assembler code must test (TST,MOVE,CMP, etc.) D0 results before branching on condition codes. Use *Scratch* by Bill Hawes (via the *scratchall* script) to catch scratch register misuse in assembler code.
- Misuse of function return values. Use function prototypes and read the Autodocs for the functions you are using. Some system functions return just success or failure, or nothing at all

(void). In such cases, the value which the function happens to return must not be used except as it is documented.

- Calling system library functions from assembler without placing the library base pointer in A6. All system functions may assume that their library's base pointer is in A6. A function's need to reference its library base can change in different OS revisions.
- Depending on unsupported side effects or undocumented behavior. Be sure to read the RKM chapters, Autodocs, and include file comments.
- Poking/peeking system private structures. Do not poke or peek any system structure unless told to do so in official Commodore documentation.
- Assuming current choices, configurations or initial values. If the current possibilities are A, B, or C, do not assume C if it isn't A or B. Check specifically for the choices currently implemented and provide default behavior for unexpected values.
- Failure to properly allocate resources before using them.
- Failure to properly close/deallocate resources.
- Improper reading/writing of hardware registers. You must mask out bits you are not interested in on reads, and write undefined bits as zero.
- Assuming initial values of hardware registers. If you are going direct to the hardware, do not depend on the initial values of any hardware registers. The settings may not be the same on different versions of the OS or from boot to boot. Always set up all of the hardware registers that affect your code. o Processor speed dependencies such as software delay loops.
- Processor instruction dependencies. Do not use instructions which are privileged on any Motorola 68xxx family processor. Do not use CLR on a hardware register which is triggered by any access (use MOVE #0 instead). The 68000 CLR instruction performs two accesses (Read, then Write). The 68020 and higher CLR instruction performs just one access.
- Depending on or failing to account for cache or prefetch effects. Self-modifying or trackdisk-loaded code requires cache flushes (see the `exec.library/CacheClearU()` function).

Amiga debugging tools such as *Enforcer*, *Mungwall* and *Scratch* can find many program bugs that may affect compatibility. A program that is *Enforcer/Mungwall/Scratch* clean stands a much better chance of working well under current and future versions of the OS. These tools are on the Denver/Milano DevCon disks. *Enforcer* and *Mungwall* are also on the kickfile disks.

2.0 Changes That Can Affect Compatibility

There are several 2.0-specific areas where OS changes and enhancements can cause compatibility problems for some software.

Exec

- Do *not* jump to location \$FC0002 as part of performing a system RESET. Many RESET functions jumped to what was the start of the ROM under 1.3. The 2.0 ROM is twice the size. We've added a *temporary* compatibility hack called "Kickety-Split" to the 2.04 Kickstart ROM. The ROM is split into two halves with a redirecting jump at \$FC0002. This hack does *not* appear on the A3000 and due to space considerations will *not* appear on future machines.
- Everything has moved.
- The Supervisor stack is not in the same place as it was under 1.3. This has caused problems for some games that completely take over the Amiga. If your program goes into Supervisor mode, you must either respect allocated memory or provide your own Supervisor stack when taking over the machine.
- ExecBase is moved to expansion memory if possible. Previously, ExecBase would only end up in one of two fixed locations. As a result, ColdCapture may be called after expansion memory has been configured. Great pains were taken to make this compatible.
- Exception/Interrupt vectors may move. This means the 68010 and above Vector Base Register (VBR) may contain a non-zero value. Poking assumed low memory vector addresses may have no effect. You must read the VBR on 68010 and above to find the base.
- No longer tolerant of wild Forbid() counts. Under 1.3, sometimes this bug could go unnoticed. Make sure that all Forbid()s are matched with one and only one Permit (and vice versa).
- When an Exec device gets an IORequest, it must validate io_Command. If the io_Command is 0 or out of range, the device must return IOERR_NOCMD and take no other action. The filesystem now sends new commands and expects older devices to properly ignore them.
- A 2.0 fix to task switching allows a busy task to properly regain the processor after an interrupt until either its quantum (4 vertical blanks) is up or a higher priority task preempts it. This can dramatically change the behavior of multitask programs where one task busy-loops while another same-priority task Wait()s. See "Task Switching" in the "Additional Information" section below.

Expansion

ExpansionBase is private—use FindConfigDev().

- Memory from contiguous cards of the same memory type is automatically merged into one memory pool.

Strap

- *romboot.library* is gone.
- *audio.device* cannot be OpenDevice()ed by a boot block program because it is not yet InitResident()ed. If OpenDevice() of *audio.device* fails during strap, you must FindResident()/InitResident() *audio.device* and then try OpenDevice() again.
- Boot from other floppies (+5,-10,-20,-30) is possible.
- Undocumented system stack and register usage at Diag and Boot time have changed.

DOS

- DOS is now written in C and assembler, not BCPL. The BCPL compiler artifact which caused D0 function results to also be in D1 is gone. 2.0 compatibility patches which return some DOS function results in both D0 and D1 are not guaranteed to remain in the next release. Fix your programs! Use *Scratch* to find these problems in your code.
- DOS now has a real library base with normal LVO vectors.
- Stack usage has all changed (variables, direction).
- New packet and lock types. Make sure you are not passing stack garbage for the second argument to `Lock()`.
- The `Process` structure is bigger. "Rolling your own" `Process` structure from a task fails.
- Unless documented otherwise, you must be a process to call DOS functions. DOS function dependence on special process structures can change with OS revisions.

Audio.device

- Now not initialized until used. This means low memory open failure is possible. Check your return values from `OpenDevice()`. This also means *audio.device* cannot be opened during 2.0 Strap unless `InitResident()`ed first. If `OpenDevice()` of *audio.device* fails during strap, you must `FindResident()/InitResident()` *audio.device*, and then try `OpenDevice()` again. There will be a small memory loss (until reboot) generated by the first opener of *audio.device* or *narrator.device* (memory used in building of *audio.device*'s base).

Gameport.device

- Initial state of hardware lines may differ.

Serial.device

- Clears `io_Device` on `CloseDevice()` (since 1.3.2)

Timer.device

- The most common mistake programmers make with *timer.device* is to send off a particular `timerequest` before the previous use of that `timerequest` has completed. Use *IO_Torture* to catch this problem.
- `IO_QUICK` requests may be deferred and be replied as documented.
- `VBLANK` timer requests, as documented, now wait at least as long as the full number of `VBlanks` you asked for. Previously, a partial vertical blank could count towards your requested number. The new behavior is more correct and matches the docs, but it can cause `VBlank` requests to now take up to 1 `VBlank` longer under 2.0 as compared to 1.3. For example, a 1/10 second request, may take 6–7 `Vblanks` instead of 5–6 `VBlanks`, or about 15% longer.

Trackdisk

- Private trackdisk structures have changed. See *trackdisk.doc* for a compatible REM-CHANGEINT.
- Buffer is freeable, so low memory open failure is possible.
- Do not disable interrupts (any of them) and then expect trackdisk to function while they are disabled.

CIA Timers

- System use of CIA timers has changed. Don't peek timers you think the system is using in a particular manner.
- Don't depend on initial values of CIA registers.
- Don't mess with CIABase. Use *cia.resource*.
- If your code requires hardware level CIA timers, allocate the timers using *cia.resource* *AddICRVector()*! Very important! Operating system usage of the CIA timers has changed. The new 2.0 *timer.device* ("Jumpy the Magic Timer Device") will try to jump to different CIAs so programs that properly allocate timers will have a better chance of getting what they want. If possible, be flexible and design your code to work with whatever timer you can successfully allocate.
- OS usage of INT6 is increasing. Do not totally take over INT6, and do not terminate the server chain if an interrupt is not for you.

Other Hardware Issues

- Battery-backed clock is different on A3000. Use *battclock.resource* to access the realtime clock if *battclock.resource* can be opened.
- A 68030 hardware characteristic causes longword-aligned longword writes to allocate a valid entry in the data cache, even if the hardware area shouldn't be cached. This can cause problems for I/O registers and shared memory devices. To solve this, either:

1. Don't do it.
 2. Flush the cache
- or
3. Use *Enforcer Quiet*.

See the Motorola 68030 manual under the description of the Write Allocate bit (which must be set for the Amiga to run with the Data Cache).

Intuition

- Private IBase variables have moved/changed. Reading them is illegal. Writing them is both illegal and dangerous.
- Poking IBase MaxMouse variables is now a no-op, but please stop poking when Intuition version is >35.
- If you are opening on the Workbench screen, be prepared to handle larger screens, new modes, new fonts, and overscan. Also see "Font" below.
- Screen TopEdge and LeftEdge may be negative.
- Left-Amiga-Select is used for dragging large screens. Do not use left-Amiga-key combinations for application command keys. The left-Amiga key is reserved for system use.
- For compatibility reasons, GetScreenData() lies if the Workbench screen is a mode only available after release 1.3. It will try to return the most sensible mode that the old OpenScreen() can open. This was necessary to keep applications that cloned the Workbench screen from having problems. To properly handle new modes, see LockPubScreen() and GetVPMODEID(), and the SA_DisplayID tag for OpenScreenTags().
- Using combined RAWKEY and VANILLAKEY now gives VANILLAKEY messages for regular keys, and RAWKEY messages for special keys (fkeys, help, etc.)
- Moving a SIMPLE_REFRESH window does not necessarily cause a REFRESHWINDOW event because layers now preserves all the bits it can.
- Sizing a SIMPLE_REFRESH window will not clear it.
- MENUVERIFY/REQVERIFY/SIZEVERIFY can time out if you take too long to ReplyMsg().
- Menu key equivalents are ignored while string gadgets are active.
- You can't type control characters into string gadgets by default. Use Ctrl-Amiga-char to type them in or use IControl Prefs to change the default behavior.
- Width and Height parameters of AutoRequest are ignored.
- New default colors, new gadget images.
- JAM1 rendering/text in border may be invisible gadgets over default colors.
- The cursor for string gadgets can no longer reside outside the cleared container area. If your gadget is, for example, 32 pixels wide with MaxChars of 4, then all 32 pixels will be cleared, instead of just 24 as was true in 1.3.
- Applications and requesters that fail to specify desired fonts will get user 2.0 Font Pref fonts that may be much larger or proportional in some cases. Screen and window titlebars (and their gadgets) will be taller when accommodating a larger font. Applications which open on the Workbench screen must adapt to variable size titlebars. Any application which accepts system defaults for its screen, window, menu, Text or IntuiText fonts must adapt to different fonts and titlebar sizes. String gadgets whose heights are too small for a font will revert to a smaller ROM font. There are now 2 different user-specifiable default system fonts which affect different Intuition features. This can lead to mismatches in mixed gadgets and text. For more information on where various system fonts come from and how they can be controlled, see "Intuition Fonts" in the "Additional Information" section below.

- Don't modify gadgets directly without first removing them from the gadget list, unless you are using a system function designed for that purpose, such as `NewModifyProp()` or `SetGadgetAttrs()`.
- Don't rely on `NewModifyProp()` to fully refresh your prop gadget after you've changed values in the structure. `NewModifyProp()` will only correctly refresh changes which were passed to it as parameters. Use `Remove/Add/RefreshGList()` for other kinds of changes.
- Custom screens must be of type `CUSTOMSCREEN` or `PUBLICSCREEN`. Other types are illegal. One application opens its screen with `NewScreen.Type = 0` (instead of `CUSTOMSCREEN`, `0x0F`). Then, when it opens its windows, it specifies `NewWindow.Type` of `0` and `NewWindow.Screen = NULL`, instead of `Type=CUSTOMSCREEN` and `Screen=(its screen)`. That happened to work before, but not anymore.
- Referencing `IntuiMessage->IAddress` as a `Gadget` pointer on non-Gadget IDCMP messages, or as a `Window` pointer (rather than looking at the proper field `IntuiMessage->IDCMPWindow`) may now cause *Enforcer* hits or crashes. The `IAddress` field used to always contain a pointer of some type even for IDCMP events for which no `IAddress` value is documented. Now, for some IDCMP events, `IAddress` may contain other data (a non-address, possibly an odd value which would crash a 68000 based system).
- Using Intuition flags in the wrong structure fields (for example, using `ACTIVEWINDOW` instead of `ACTIVATE`). To alleviate this problem, 2.0 has introduced modern synonyms that are less confusing than the old ones. For example, `IDCMP_ACTIVEWINDOW` and `WFLG_ACTIVATE`. This particular example of confusion (there are several) was the nastiest, because `IDCMP_ACTIVEWINDOW`, when stuffed into `NewWindow.Flags`, corresponds numerically to `WFLG_NW_EXTENDED`, which informs Intuition that the `NewWindow` structure is immediately followed by a `TagList`, which isn't there! Intuition does some validation on the `Taglist` pointer in order to partially compensate. To make your compiler use the new synonyms only, add this line to your code before Intuition include files:
 - `#define INTUL_V36_NAMES_ONLY.`
- Do not place spaces into the `StringInfo->Buffer` of a `LONGINT` string gadget. Under 1.3, it turned out that worked, but under 2.0, the validation routine that checks for illegal keystrokes looks at the contents for illegal (i.e., nonnumeric) characters, and if any are found assumes that the user typed an illegal keystroke. The user's only options may be shift-delete or Amiga-X. Use the correct justification instead.
- If you specify `NULL` for a font in an `IntuiText`, don't assume you'll get Topaz 8. Either explicitly supply the font you need or be prepared to size accordingly. Otherwise, your rendering will be wrong, and the user will have to reset his Preferences just to make your software work right.
- Window borders are now drawn in the screen's `DetailPen` and `BlockPen` rather than the window's pens. For best appearance, you should pass an `SA_Pens` array to `OpenScreen()`. This can be done in a backwards compatible manner with the `ExtNewScreen` structure and the `NS_EXTENDED` flag.
- The system now renders into the full width of window borders, although the widths themselves are unchanged.
- Window borders are filled upon activation and inactivation.

- Window border rendering has changed significantly for 2.0. Note that the border dimensions are unchanged from 1.x (Look at `window->BorderLeft/Top/Width/Height` if you don't believe it!). If your gadget intersects the border area, although it may have looked OK under 1.3, a visual conflict may occur under 2.0. If Intuition notices a gadget which is substantially in the border but not declared as such, it treats it as though it is in the border (this is called "bordersniffing"). Never rely on Intuition to sniff these out for you; always declare them explicitly (see the Gadget Activation flags `GACT_RIGHTBORDER`, etc.). See "Intuition Gadgets" and "Window Borders" in the "Additional Information" section below.

Preferences

- Some old Preferences structure fields are now ignored by *SetPrefs* (e.g., `FontHeight`). *SetPrefs* also stops listening to the pointer fields as soon as a new-style pointer is passed to Intuition (new-style pointers can be taller or deeper).
- Preferences `ViewX/YOffset` only applies to the default mode. You cannot use these fields to move the position of all modes.
- The Preferences `LACEWB` bit is not necessarily correct when Workbench is in a new display mode (akin to `GetScreenData()`).

Workbench

- The Workbench GUI now has new screen sizes, screen top/left offsets, depths, modes, and fonts.
- Default Tool now searches paths.
- New Look (boxed) icons take more space.
- Do not use icons which have more 1-bits set in `PlanePick` than planes in the `ImageData` (one IFF-to-Icon utility does this). Such icons will appear trashed on deeper Workbenches.
- New Look colors have black and white swapped (as compared to 1.3).
- The Workbench screen may not be open at startup-sequence time until some output occurs to the initial Shell window. This can break startup-sequence-started games that think they can steal WB's screen bitplanes. Do not steal the WB screen's planes (For compatibility, booting off pre-2.0 disks forces the initial screen open. This is not guaranteed to remain in the system). Use startup code that can detach when run (such as `cback.o`) and use `CloseWorkbench()` to regain the screen's memory.

Under 1.3 the Workbench Screen and initial CLI opened before the first line in `s:startup-sequence`. Some naughty programmers, in an attempt to recover memory, would search for the bitplane pointers and appropriate the memory for their own use. This behavior is highly unsafe.

By default 2.0 opens the initial CLI on the first `_output_` from the `s:startup-sequence`. This allows screen modes and other parameters to be set before the user sees the screen. However, this broke so many programs that we put in the "silent-startup" hack. A disk installed with 1.3 install opens the screen as before. A disk installed under 2.0 opens silently. Never steal the Workbench bitplanes. You don't know where they are, how big they are, what format they may be in, or even if they are allocated. Recovering the memory is a bit tricky.

Under 2.0

Simply avoid any output from your s:startup-sequence. If your program opens a screen it will be the first screen the user ever sees. Note that if ENDCLI is ever hit, the screen will pop open.

Under 1.3

After ENDCLI, call the `CloseWorkbench()` function to close the screen. This also works under 2.0. Loop on `CloseWorkbench()` with a delay between loops. Continue looping until `CloseWorkbench()` succeeds or too much time has passed. Note that a new program called *EndRun* is available for starting non-returning programs from the startup-sequence. *EndRun* will reduce memory fragmentation and will close Workbench if it is open. *EndRun.lzh* will be available in Commodore's Amiga listings area on BIX.

Layers

- Use `NewLayerInfo()` to create, not `FattenLayerInfo()`, `ThinLayerInfo()` or `InitLayers()`.
- Simple-refresh preserves all of the pixels it can. Sizing a `SIMPLE_REFRESH` window no longer clears the whole window.
- Speed of layer operations is different. Don't depend on layer operations to finish before or after other asynchronous actions.

Graphics

- Do not rely on the order of Copper list instructions. For example, 2.0's `MrgCop()` builds different Copper lists to that of 1.3, by including new registers in the list (e.g., `MOVE xxxx,DIWHIGH`). This changes the positions of the other instructions. We know of one game that "assumes" the `BPLxPTRs` would be at a certain offset in the Copper list, and that is now broken on machines running 2.0 with the new Denise chip.
- Graphics and layers functions which use the blitter generally return after *starting* the final blit. If you are mixing graphics rendering calls and processor access of the same memory, you must `WaitBlit()` before touching (or deallocating) the source or destination memory with the processor. For example, the `Text()` function was sped up for 2.0, causing some programs to trash partial lines of text.
- `ColorMap` structure is bigger. Programs must use `GetColorMap()` to create one.
- Blitter runs decide ascend/descend on 1st plane only.
- Changing the display mode of an existing screen or viewport while it is open is still not a supported operation.
- `GfxBase DisplayFlags` and row/cols may not match Workbench screen.
- Do not hardcode modulo values—use `BitMap->BytesPerRow`.
- If the graphics *Autodocs* say that you need a `TmpRas` of a certain size for some functions, then you must make that the minimum size. In some cases before 2.0, you may have gotten away with using a smaller `TmpRas` with some functions (for example `Flood()`). To be more robust, Graphics now checks the `TmpRas` size and will fail the function call if the `TmpRas` is too small.

- ECS chips under 2.0 use a different method of generating displays. The display window registers now control DMA.
- **LoadRGB4()** used to poke colors into the active copperlist with no protection against deallocation of that copperlist while it was being poked. Under 2.0, semaphore protection of the copperlist was added to **LoadRGB4()**. This semaphore protection makes it totally incorrect and extremely dangerous to call **LoadRGB4()** during an interrupt. The general symptom of this problem is that a system deadlock can be caused by dragging one screen up and down while another is cycling. Color cycling should be performed from within a task, not an interrupt. Note that in general, the only functions which may be safely called from within an interrupt are the small list of Exec functions documented in the "Exec: Interrupts" chapter of the 1.3 *Amiga ROM Kernel Manual: Libraries and Devices*.

Fonts

- Some font format changes (old format supported).
- Private format of .font files has changed (use **FixFonts** to create).
- Default fonts may be larger, proportional.
- Topaz is now sans-serif.
- Any size font will be created via scaling as long as **TextAttr.Flags FPF_DESIGNED** bit is not set. If you were asking for an extreme size, like size 1 to get the smallest available, or 999 to get the largest available, you will get a big (or very, very small) surprise now.
- Do not use -1 for **TextAttr.Flags** or **TextAttr.Styles**, nor as the flags for **AvailFonts()** (one high bit now causes **AvailFonts()** to return different structures). Only set what you know you want. A kludge has been added to the OS to protect applications which currently pass -1 for **AvailFonts()** flags.

CLI/Shell

- Many more commands are now built-in (no longer in C:). This can break installation scripts that copy C:commandname, and programs that try to **Lock()** or **Open()** C:commandname to check for the command's existence.
- The limit of 20 CLI processes is gone and the **DOSBase CLI** table has changed to accommodate this. Under V36 and higher, you should use the new 2.0 functions rather than accessing the CLI table directly.
- Shell windows now have Close Gadgets. The EOF character is passed for the Close Gadget of a Shell. This is -1L with **CON: getchar()**, and the Close Gadget raw event ESC seq with **RAW:.**
- Shells now use the simple-refresh character-mapped console (see "Console" below).

Console

- By default, CON: now opens SIMPLE_REFRESH windows using the V36/V37 console character mapped mode. Because of some differences between character mapped consoles and SMART_REFRESH non-mapped consoles, this may cause incompatibilities with some applications. For example, the Amiga private sequences to set left/top offset, and set line/page length behave differently in character mapped console windows. The only known workaround is to recompile asking for a CON: (or RAW:) window using the SMART flag.
- Simple refresh/character mapped console windows now support the ability to highlight and copy text with the mouse. This feature, as well as pasting text should be transparent to programs which use CON: for console input and output. Pasted text will appear in your input stream as if the user had typed it.
- While *Conclip* (see s:startup-sequence) is running, programs may receive "<CSI>0 v" in their input stream indicating the user wants to paste text from the clipboard. This shouldn't cause any problems for programs which parse correctly (however we know that it does; the most common problems are outputting the sequence, or confusing it with another sequence like that for FKEY 1 which is "<CSI>0 ").
- The *console.device* now renders a ghosted cursor in inactive console windows (both SMART_REFRESH, and SIMPLE_REFRESH with character maps). Therefore, rendering over the console's cursor with *graphics.library* calls can trash the cursor; if you must do this, first turn off the cursor.
- Some degree of unofficial support has been put in for programs which use SMART_REFRESH console windows, and use *graphics.library* calls mixed with *console.device* sequences to scroll, draw text, clear, etc. This is not supported in SIMPLE_REFRESH windows with character maps, and is strongly discouraged in all cases.
- Closing an Intuition window before closing the attached *console.device* worked in the past; it will now crash or hang the machine.
- Under 1.2–1.3, vacated portions of a console window (e.g., areas vacated because of a clear or a scroll) were filled in with the character cell color. As of V36 this is no longer true, vacated areas are filled in with the global background color which can be set using the SGR sequence "<ESC>[>##m" where ## is a value between 0–7. In order to set the background color under V36/V37, send the SGR to set background color, and a FORMFEED to clear the screen.
- Note that SIMPLE_REFRESH character mapped consoles are immediately redrawn with the global background color when changed—this is not possible with SMART_REFRESH windows.

Additional Information

Task Switching

The 1.3 Kickstart contained two task switching bugs. After an interrupt, a task could lose the CPU to another equal priority task, even if the first task's time was not up. The second bug allowed a task whose time was up to hold on to the CPU either forever or until a higher priority task was scheduled. Two busy-waiting tasks at high priority would never share the CPU. Because the *input.device* runs at priority 20, usually the effect of these bugs was masked out for low priority tasks. Because of the bugs, the ExecBase->Quantum field had little effect.

For 2.0, a task runs until either its Quantum is up, or a higher priority task preempts it. When the Quantum time is up, the task will now lose the CPU. The Quantum was set to 16/60 second for 1.3, and 4/60 second for 2.0.

In general, the 2.0 change makes the system more efficient by eliminating unnecessary task switches on interrupt-busy systems (for example, during serial input). However, the change has caused problems for some programs that use two tasks of equal priority, one busy-waiting and one Wait()ing on events such as serial input. Previously, each incoming serial character interrupt would cause task context switch allowing the event handling task to run immediately. Under 2.0 the two tasks share the processor fairly.

Intuition Gadgets and Window Borders

If 2.0 Intuition finds a gadget whose hit area (gadget Left/Top/ Width/Height) is substantially inside the border, it will be treated as though it was declared in the border. This is called "bordersniffing". Gadgets declared as being in the border or detected by Intuition as being in the border are refreshed each time after the border is refreshed, and thus aren't clobbered.

Noteworthy special cases:

1. A gadget that has several pixels not in the border is not bordersniffed. An example would be an 18-pixel high gadget in the bottom border of a SIZEBBOTTOM window. About half the gadget will be clobbered by the border rendering.
2. A gadget that is not substantially in the border but has imagery that extends into the border cannot be sniffed out by Intuition.
3. A gadget that is substantially in the border but has imagery that extends into the main part of the window will be sniffed out as a border gadget, and this could change the refreshing results. A common trick to put imagery in a window is to put a 1x1 or 0x0 dummy gadget at window location (0,0) and attach the window imagery to it. To support this, Intuition will never bordersniff gadgets of size 1x1 or smaller.

All these cases can be fixed by setting the appropriate GACT_xxxBORDER gadget Activation flag.

4. In rare cases, buttons rendered with Border structures and JAM1 text may appear invisible under 2.0. We apologize, but there is nothing that can be done on our end, even if the application technically did nothing wrong.

Intuition Fonts

The following table shows where the Intuition gets its fonts from:

What you tell OpenScreen	Screen's Font	Window's RPort's Font
A. NewScreen.Font = myfont	myfont	myfont
B. NewScreen.Font = NULL	GfxBase->DefaultFont	GfxBase->DefaultFont
C. {SA_Font, myfont}	myfont	myfont
D. {SA_SysFont, 0}	GfxBase->DefaultFont	GfxBase->DefaultFont
E. {SA_SysFont, 1}	Font Prefs Screen text	GfxBase->DefaultFont

Notes:

A and B are the options that existed in releases prior to V36.

C and D are new V36 tags that are equivalent to A and B respectively.

E is a NEW option for V36. The Workbench screen uses this option.

GfxBase->DefaultFont will always be monospace. This is the "System Default Text" from Font Preferences.

The "Screen Text" choice from Font Preferences can be monospace or proportional.

"myfont" can be any font of the programmer's choosing, including a proportional one. This is true under all releases of the OS.

The menu bar, window titles, menu items, and the contents of a string gadget use the screen's font. The font used for menu items can be overridden in the item's **IntuiText** structure. Under V36 and higher, the font used in a string gadget can be overridden through the **StringExtend** structure. The font of the menu bar and window titles cannot be overridden. Because the 2.0 Workbench screen uses option E to specify its Screen font from the user's Screen font Preferences, applications which open windows on the Workbench screen may get very large or proportional fonts in their menu bars, window titles, menu items and string gadgets.

To predict your window's titlebar height before you call **OpenWindow()**:

```
topborder = screen->WBotTop + screen->Font->ta_YSize + 1
```

The screen's font may not legally be changed after a screen is opened.

Be sure the screen cannot go away on you. This is true if:

1. You opened the screen yourself.
2. You currently have a window open on the screen.
3. You currently hold a lock on this screen (see **LockPubScreen()**).

IntuiText rendered into a window (either through **PrintIText()** or as a gadget's **GadgetText**) defaults to the Window **RastPort** font, but can be overridden using its **ITextFont** field. Text rendered with the **Text()** graphics.library call appears in the Window **RastPort** font.

The Window's **RPort's** font shown above is the initial font that Intuition sets for you in your window's **RastPort**. It is legal to change that subsequently with **SetFont()**.

CDTV Feature Overview

While CDTV has all of the features of the standard Amiga (with the notable exception of a standard alphanumeric keyboard and a mouse), CDTV has many additional features not found in the standard Amiga. The following is a description of some of the more interesting features of the standard CDTV.

cdtv.device

The *cdtv.device* is a standard Amiga Exec device that provides access to and control of the CD-ROM mechanism. The *cdtv.device* allows control of CD audio, access to CD-ROM data disks, and will automatically handle mixed mode discs containing both CD audio and data. By sending standard Amiga I/O Requests to the *cdtv.device*, the application can read CD-ROM data, play CD Digital Audio (CD-DA), track the position of the laser, synchronize audio with other events, read the CD Table of Contents, set CD audio output attenuation level, read the status of the CD drive, read error conditions, and enable/disable the front panel controls of the CDTV.

bookmark.device

The *bookmark.device* is a standard Amiga Exec device which provides access to and control of the non-volatile RAM of the CDTV unit. Bookmarks provide a means of storing CDTV application data and preserving it across machine resets and power button shutdowns. For now, bookmarks (and cardmarks, see below) will be the only semi-permanent data storage method available to CDTV developers until floppy drives and SCSI hard disks become widespread.

Bookmarks are designed primarily to hold small amounts of data that can be used to return the user to a previously marked state within an application, hence the term bookmark. A bookmark is internally identified by means of a Manufacturer ID (assigned by CATS or the Special Projects Group) and a Product Code (assigned by the software developer). This enables applications to easily distinguish their bookmarks from other bookmarks. The *bookmark.device* provides a means of defining priorities of the various bookmarks, and for aging bookmarks so that unused bookmarks will be the first to be replaced as the bookmark space fills and additional space is required.

Currently, CDTV machines are equipped with 2K of bookmark memory. This will most likely change as CDTV evolves, but using the *bookmark.device* will shield developers from changes in the amount, type, or location of this RAM.

cardmark.device

The *cardmark.device* is virtually identical to the *bookmark.device* except that it provides access to and control of credit card style memory cards instead of the non-volatile RAM. These cards, which plug into the front of the CDTV unit, may also be used for ROM Operating System enhancements such as new libraries or devices, ROM application programs, expansion ram for CDTV main memory, recoverable ram disks, diagnostic software, and special hardware enhancements in addition to being used for cardmarks.

playerprefs.library

The *playerprefs.library* is a standard Exec library that contains routines to assist an application in CDTV specific user interface areas. The *playerprefs.library* gives access to the CDTV Preferences stored in the non-volatile RAM, provides screen centering routines, bitmap manipulation routines, and infrared (IR) controller input handling routines, as well as a screen blanker. Use of the routines in this library will provide some consistency between applications, and will enable an application to easily obey the user preferences choices.

ISO 9660 filesystem

The ISO 9660 Filesystem provides an Amiga application with transparent access to a CD-ROM. In most respects, the application can treat the disc as an Amiga standard disk with a large amount of storage. This makes developing CDTV applications much easier than if a different access method had been used. (There are some important differences, however, that must be kept in mind to achieve maximum performance).

CDXL

CDXL is a technique incorporated in the CDTV device driver that can significantly improve the transfer speed of data from the CD disc into the CDTV player. It is not a compression method nor is it a way of speeding up the drive. Instead, CDXL is a method of arranging the data on the CD-ROM such that seek times are minimized. The CDXL routines allow an application to quickly locate the data it needs and move it from the disc into memory in a minimum amount of time.

When using CDXL transfer routines, most of the processor time is still available for application use because CDXL itself only uses about 8% of the capacity of the 68000 processor. Having most of the resources of CDTV available allows applications to do interesting things while the data transfer is taking place. Though the main use to date for CDXL has been for display of 1/3 screen images running at about 12 frames per second with audio, the CDXL technique is not limited to this. It can be used with any type of data—programs, audio, images, etc. When CDXL is in use, the CDTV disc light remains lit showing the maximum data transfer rate has been achieved.

PORTS

Centronics Parallel Port

This is an industry standard Centronics parallel port to connect printers.

RS232 Serial Port

This is a standard RS232 port for connection of serial printers, modems, and other RS232 accessories.

RAM/ROM Card Slot

Expansion RAM credit cards, ROM cards, and special hardware expansion cards can be plugged into this port on the front of the CDTV unit.

Video Expansion Slot

Normally contains a card with the following connectors:

- Composite video (RCA connector).
- S-VHS (S connector).
- RF modulated (F type connector).

Other video cards are possible, such as the optional genlock card. This is a software controllable genlock module that allows mixing CDTV video with external video input.

DMA Expansion Port

This port can have a SCSI card, a LAN card, or other cards which require a DMA interface to the main CDTV memory.

Floppy Port (Amiga compatible)

A standard Amiga floppy or a CDTV black, matching floppy can be plugged into this port. This will allow CDTV applications to read and write standard Amiga 880K floppy disks.

Audio Connectors

Two stereo output jacks (RCA connectors).

One stereo headphone jack.

MIDI Ports

There are two MIDI ports:

- MIDI In.
- MIDI Out.

Keyboard Port

An optional CDTV wired keyboard may be plugged into this port. The addition of both the keyboard and the external floppy will allow the CDTV unit to be used as an Amiga computer as well as a CDTV.

Mouse Port

An optional wired mouse may be plugged in into this port.

Booting A CDTV Application

Starting Your Application

Applications on CDTV are generally “self-starting”. Unlike programs that run on the normal Amiga where a program is normally invoked by the user from either Workbench or the Shell, on CDTV, you are almost always going to be invoked directly from the startup-sequence on your CD-ROM disc.

The normal purpose of a startup-sequence is to set up the environment of the computer. In the case of a CDTV application, your application will also be invoked. On the Amiga, many users spend much time tuning their own startup-sequence files; on CDTV, you, the developer, are the one who will spend the time.

The CDTV startup-sequence must start your application as soon as possible. Users are not going to be satisfied watching a blank screen. They want something to happen as soon as they turn the power on. In this, we need to try to match the boot up speed of a typical game console. While we can't quite get there, we can get pretty close.

The goal for your startup-sequence is to start your application in the correct environment; your constraints should be to bring up your application as quickly as possible, and to avoid fragmenting memory while doing so.

There are several issues to starting quickly. One of the most important is that the perceived time before something happens needs to be as short as possible.

This may involve showing a picture during the setup of your application or playing some music. However, the time needed to load and start one of these subprocesses must be carefully weighed against the result since displaying a picture will take additional time.

Some General Rules

- No disk thrashing.
- Minimize the number of programs invoked in your startup-sequence.
- Use the proper commands in your startup-sequence.
- Test your startup-sequence.
- Try to cut the perceived start time down as much as possible.
- Use *QuickStart*.

Specific Rules

Rule 1: Multitasking Is Not Always The Answer

Even though the Amiga is a multitasking computer, there are some places where it pays to do one thing at a time. Loading files from disk is one of them. When you attempt to load two files at a time, the filesystem ends up interleaving block requests, which produces a lot of seeks. (On CDTV, seeks should be avoided at all costs; seeking is very expensive in time). The load time becomes much longer than just the sum of the two load operations.

What not to do

Under 1.3, the *Run* command loads its target command asynchronously. This means if you put the following in your startup-sequence:

```
run display picture
start_app
```

Run is loaded and started. Then *Run* begins to load the *Display* program. Meanwhile, the *Start_app* program load begins. We get immediate disk thrashing, as we attempt to load both *Display* and *Start_app* at the same time. (Under 2.0 this problem does not exist as the *Run* command waits until it has finished its load operation before allowing the script to continue. However, CDTV must always consider the 1.3 environment.)

Basically, you should avoid the use of the *Run* command in starting your CDTV application from the startup-sequence if you want to avoid disk thrashing—which you do.

Rule 2: A Short Startup-sequence Is Best

Keep your startup-sequence as short as possible. Just perform the minimum number of things needed to get your application up and running. Combining functions into larger, multipurpose programs can help a lot. For instance, if you use the *playerprefs.library*, you can probably eliminate your use of *Bookit* to read the CDTV preferences.

Rule 3: The Right System-Configuration Makes All The Difference

The proper system-configuration file in *devs:* will make a difference. At the minimum, you will want to specify both the default screen color palette and the default pointer color palette to be all black. This will avoid any unsightly display on booting.

You should also make other reasonable default settings. Remember, only some of the Preferences settings are controlled by the user in CDTV Player Preferences. For the rest, the user is depending on you to choose a good set of defaults for the particular application ...since the user is booting from your CD-ROM, you have control over the settings. Always test these actual settings with your application to make sure they are appropriate.

Printers Drivers. If your application includes printing, the printer drivers are going to have to be included in the *CD0:devs/printers* directory, as well as having the *printer.device*, *parallel.device*, and *serial.device* present.

Rule 4: Five Commands Can Improve The Environment

The standard CDTV utility commands you will be most interested in for setting up the environment of your application via the Startup-Sequence are *Bookit*, *CDSetMap*, *EndRUN*, *Keeper*, and *RMTM*.

Bookit

Bookit allows you to read the CDTV Preferences settings stored in the non-volatile RAM. While the same functions can be performed directly from your program, *BookIt* provides a convenient, externally controllable method of obeying the settings.

CDSetMap

CDSetMap will set the default system keymap to a selection based on the language selection the user has made in CDTV Preferences.

Keeper

Keeper displays a picture while the rest of your application loads; when your application starts, *Keeper* will remove the picture.

EndRUN

EndRUN launches a program in the background after closing the CLI from which the command was run, as well as the Workbench. This will free up additional memory for use by the application.

RMTM

RMTM removes the trademark screen. This is required if you wish users to actually see your application.

The full documentation for these five utilities is at the end of this article.

Always end your startup-sequence with an *Endcli*. This should never get executed, but just in case your application accidentally exits, this will prevent the user from getting into a CLI where he could be lost forever—especially without a keyboard)

Rule 5: Test Now, Avoid Problems Later

Test your startup-sequence. Unless you are one of the lucky ones who is using the CDTV emulator card to develop, the first real test of your application setup will be when you press a gold disk. It's always a shame to have to redo a gold disk, especially when the only problem is that you don't have the proper assign in place.

When you start your application from the CD-ROM, the CD can be referred to as either SYS: or CD0: You should always refer to it by one of those names. Getting tricky with assigns is usually not worth it. During testing on CDTV when you are booting from a floppy, you can refer to anything on the floppy as SYS: and any data as CD0:. During testing on your development machine from a hard disk, you can do the same if you make a simple assign of CD0: to the partition on your hard disk where you keep the data you will put on your CD-ROM. This will allow you to use the same assign set for development, testing, and final product. This can cut down on a common source of mistakes that could cause you to have to remake a gold disk.

Rule 6: Don't Allow Users To Strum Their Fingers

Cut down on perceived startup time as much as possible. The amount of time the user thinks it takes before your application starts is at least as important as the actual time it takes. Display of a title screen, a picture, or simple animation, or playing some music can take the user's mind off the wait, and make it seem that your application is running, when in reality it still has another twenty-five seconds of loading to do.

It's about three seconds from the time the system is reset to the time it begins reading your startup-sequence. There's not a lot you can do about this time—much of it is caused by having to wait for the CD-ROM hardware to settle down.

The *Keeper* utility can be used to load an IFF picture before your main application loads. It will remain in the background. When your application is ready to begin, it can signal *Keeper* to remove the picture.

Rule 7: Use QuickStart

While researching this article, it became apparent that some quicker method using the direct read capability of the *cdtv.device* would really help speed up start times. *QuickStart* is the result. Basically, *QuickStart* is a combined direct read file loader/simple unarchiver that uses Burst read.

QuickStart comes in two parts: a packer and an unpacker. Using the packer, you can combine all the programs invoked in your startup-sequence into one big file with a script to run them. Then you can use the unpacker to load them out into RAM:, where you can execute them in any order, run them in the background, or whatever. After your application has started, it can delete the commands in RAM: to regain the memory space used during the *QuickStart* process, so except for the overhead for the RAM-handler (10K under 1.3), there is only a small loss of total memory.

MEMORY FRAGMENTATION ISSUES DURING STARTUP

Pay attention to fragmentation during startup. Your goal is to have as big a contiguous RAM area as possible for your application. The order in which you do things can be critical.

For instance, always create the RAM disk by *CDing* to it, rather than copying. Reason? The order of memory allocation. If you invoke by *Copy* the order goes:

Copy allocates a buffer to hold your file. Then looks at the destination. Poof, the RAM-handler is invoked, and it allocates its memory. *Copy* finishes, freeing its buffers, but since RAM: started after *Copy*, you now have a free space, the RAM-handler, and free space.

To avoid this, *CD* to RAM: first. This will invoke the RAM-handler. Then the order is RAM is invoked, allocating its memory. Then *Copy* does its allocation, and frees its allocation. Now you still have basically the same RAM configuration you began with. (You will have a small fragmentation, as the *CD* command needed to be loaded; under 2.0, where the *CD* command is built into the shell, you will not have this fragmentation)

Documentation On CDTV Startup-sequence Utilities

NAME

Bookit—CDTV configuration/bookmark reader utility

SYNOPSIS

Bookit [*bjvcs1*]

FUNCTION

This program configures CDTV options based on the contents of the CDTVPrefs bookmark, stored in non-volatile RAM. This program ideally should be part of the Startup-sequence. The options are:

- b** Sets Workbench screen and pointer colors to black. This is helpful in presenting a blank screen after taking down the CDTV trademark, rather than the jarring blue Workbench screen.

NOTE:

Workbench must be the only screen open for this to operate properly.

- j** Installs the Joy/Mouse rerouting task, which takes joystick events on gameport 1 and feeds them down the input.device food chain as mouse events. This makes applications immune to the state of the IR remote's JOY/MOUSE button.

For more details, see the autodoc `playerprefs.library/InstallJoyMouse`.

- v** Centers the Intuition View based on the user's centering stored in the bookmark. It is highly recommended that all CDTV titles make use of this option.
- c** Starts the system's key click task with default settings. This will cause a small "BEEP!" to be emitted each time the user presses a key or mouse button.

For more details, see the autodocs `playerprefs.library/InstallKeyClick` and `playerprefs.library/KeyClickCommand`

- s** Starts the system's screen saver task with default settings. The time delay is taken from the CDTVPrefs bookmark.

For more details, see the autodocs `playerprefs.library/InstallScreenSaver` and `playerprefs.library/ScreenSaverCommand`

- l** Sets the LACE bit in `GfxBase.system_bplcon0`. This will cause all Views to come up interlaced. This is useful for recording more reliably to VCRs. This operation is identical to the old PD program 'SetLace.'

NOTES

The options 'c,' 's,' and 'l' will have no effect if the user has not enabled the corresponding item in CDTVPrefs. We strongly recommend all CDTV titles make use of the 'c,' 's,' and 'l' options which will guarantee configuring the system to the users preferences.

EXAMPLE

```
Bookit cslv ; Recommended defaults
Bookit cslvj ; Defaults plus joy/mouse rerouter
```

BUGS**SEE ALSO**

playerprefs.library/InstallJoyMouse
playerprefs.library/RemoveJoyMouse
playerprefs.library/InstallKeyClick
playerprefs.library/KeyClickCommand
playerprefs.library/InstallScreenSaver
playerprefs.library/ScreenSaverCommand

NAME**EndRun****SYNOPSIS**

EndRun [command]—Execute the command given with workbench closed

FUNCTION

This CLI-only command can be used in the startup-sequence to execute the command given after the workbench screen has been closed down. If running under 2.0 and the workbench screen has not opened yet, it will just execute the command.

INPUTS

[command] is the command line to be executed. The first word in the command will be the load file as passed to LoadSeg(). If it does not exist, EndRun will just halt.

RESULTS

The command given is run with the workbench screen closed.

EXAMPLE

```
; startup-sequence for SnakePit
EndRun SnakePit datafile
; Note that we run the snakepit program with the
; argument "datafile" which is passed to the program.
```

NOTES

If run under 2.0, this may end up doing almost nothing except prevent the opening of the Workbench screen.

The [command] will not have any functioning stdin/stdout. stdin/stdout will be connected to NIL:

EndRUN ***MUST*** have V1.2 or greater kickstart. It also ***MUST*** be executed from a CLI. (It does not check for workbench)

EndRun has a maximum COMMAND name (first word in the command line, including the complete path) of 64 characters. It will not work correctly with more. The rest of the command line can be as long as you wish.

When the program exits, Endrun will just halt. It will go into a dead loop. However, applications run with EndRun are usually of the type the user uses with a reboot as this utility is design for use in startup sequences.

BUGS

The command invoked by EndRUN must have the full path specified, or the endrun process will not find the command, and hang.

NAME

Keeper— CDTV IFF display utility

SYNOPSIS

Keeper [filename] [QUIT]

FUNCTION

This program is used to display an IFF ILBM picture as a background task while the rest of an application boots. Keeper can be used in the Startup-sequence of a CDTV application to load a picture. This picture will be displayed until it is signaled either by the application or by another invocation of keeper in the Startup-Sequence that it is time to remove the picture.

The QUIT option will cause Keeper to signal the background keeper task to shut down. This may also be done directly from the application. The name of the Keeper task is "PicKeeper". By signaling it with a SIGBREAKF_CTRL_C, the keeper task is notified of the shutdown.

Keeper can display any standard IFF picture including overscan except for pictures requiring custom copper list changes. Keeper automatically compensates for PAL by centering its View appropriately using the CDTV player preferences. If the CDTV preferences have not been set, Keeper will use the information from the active View.

BUGS

The Keeper background task periodically checks the GfxBase->ActiView variable to see if its View has been replaced. Because of this it is dangerous for a program to save the active View during initialization and restore it later. CloseWorkBench() and OpenWorkBench() should be used instead.

SEE ALSO

NAME

RMTM

SYNOPSIS

RMTM

FUNCTION

This utility removes the CDTV trademark screen. It must be run before a CDTV application starts; otherwise the CDTV application will remain covered by the CDTV trademark logo.

BUGS

SEE ALSO

NAME

MakeQuick

SYNOPSIS

MakeQuick [FROM/M] [TO/K]

FUNCTION

MakeQuick creates an archive file containing all the files specified for use with the **QuickStart** utility. The default archive name is 'quickfile'. To select a different archive name, the **TO** keyword must be supplied. The archive can contain executables, scripts, data files, etc. No compression on the archive is done.

BUGS**SEE ALSO**

QuickStart

NAME**QuickStart****SYNOPSIS****QuickStart [FROM] [TO/K] [BURST/S]****FUNCTION**

The QuickStart utility is used to read an archive created by the MakeQuick command, and unpack it into the target directory. The default archive file name is 'quickfile'. The default TO directory is the current directory. The BURST keyword will allow the QuickStart command to load the archive using a DIRECT READ when loading from a CD disc on a Commodore CDTV.

After unpacking the archive, if there is a script file named 'autostart' it is executed.

NOTES

After the autoscript command is run, the application should delete any unneeded commands (including itself) to regain RAM space.

Future changes to the CDTV system may remove the need for QuickStart. A method will be provided to detect this.

BUGS

If a subdirectory is not present, QuickStart will not create it.

Localization Programming

The CDTV player is sold worldwide. Its Preferences contain fifteen languages. When preparing a CDTV title for publication, you should consider making the title suitable for as many markets as possible. This will require a certain amount of effort on your part, but the rewards can be enormous.

CDTV is a consumer product. It is sold through mass merchandisers, and is aimed at the most general audience possible. While many computer users manage to survive using foreign-language (read: *English*) versions of software, you cannot assume the average consumer will be as patient.

Many potential buyers will not even consider a title unless that title is in their language. Some countries (such as France) have laws requiring products to include instructions in the local language.

See the article, "Localizing CDTV and CDTV Applications" in the User Interface section of this reference manual for general information on how to localize your product.

Localization is the process by which software dynamically adapts to different locales. A locale is made up of a set of attributes describing a language, and a set of cultural and linguistic conventions. Without a standardized method to handle localization, the task of localizing applications is significant, as either there must be several versions of the application, or the application must adapt on the fly. Full localization goes far beyond the simple selection of the appropriate keyboard keymap for use in a particular country. A fully localized application will appear as if it were written only for the country for which its locale is set.

Under V2.1 of the Amiga operating system, there will be an Amiga shared library called the *locale.library* which has functions for displaying, formatting, managing multiple translations of text strings within an application, string sorting, currency symbol selection and formatting, etc. Unfortunately, CDTV applications cannot use the features of V2.1 at this time because the current version of the *locale.library* depends heavily on other features of the V2.1 operating system.

A CDTV application must consider V1.3, as the majority (close enough to 100% to make no difference) of CDTV units in the field are V1.3. While there will be V2.0 CDTV machines (like the A690 attached to a V2.0 equipped A500), the CDTV universe is really a V1.3 one, at least for the time being. This means the CDTV application is on its own for localization services.

Techniques For Creating Multilingual Applications

Localizing a product requires careful planning. The following items, if designed into the project from the beginning, may simplify things when you begin translating and adapting your product.

Multilingual applications should read the language preference set by the user. This is done by using the *playerprefs.library* to read the CDTV Preferences stored in the non-volatile RAM. There are fifteen languages currently supported. (This is subject to change, as the CDTV market spreads elsewhere in the world). The currently supported languages are:

CDTV Preferences Language Choices

American English	English	German	French
Spanish	Italian	Portuguese	Danish
Dutch	Norwegian	Finnish	Swedish
Japanese	Chinese	Korean	

There's More Than Meets The Eye. Although the CDTV Preferences only lists fifteen language selections, eighteen languages are currently defined in *cdtvprefs.h* as supportable languages. The three additional languages are Greek, Arabic and Hebrew.

As language selection is a part of CDTV Player Preferences, it is not usually necessary for your title to include its own language selection screen unless you can support additional languages than those listed above.

The *getprefs.c* program in the system directory of the CDTV Tools Disk V1.1 shows an example of how to read the Preferences setting in your application.

If your application does not support the preferred language, you should propose a list of the languages you do support, and let the user select an alternate language.

Methods Of Organizing Localized Text Strings

Due to the large amount of data storage available on the CD-ROM disc, it is actually practical to store multiple versions of the application on the disc, and to load the appropriate version based on the language selected. This presents the most straightforward programming model for the various languages. If this method is adopted, it is strongly suggested that you keep a unified source tree, and conditionally compile in the various sets of language strings. Again, the large amount of space available on the CD-ROM disc makes this method quite practical. It also keeps RAM usage down, as the language selection code does not have to remain resident in RAM, and the screen formatting routines can be simpler.

While it may be necessary to design several versions of the same screen to accomodate the differing lengths of text strings in different languages, it will not be necessary to design dynamic screen layout engines, a much more complex task. Making the application completely font and text length sensitive is certainly possible and should result in a good looking screen layout, but may be more work than is necessary.

Another, more simple method to handle the screen layout problem is to design the screen layout so that there is always enough room for the largest translated text string in each place a text string is used. While this method does tend to spread control icons apart, it is not necessarily a drawback for a CDTV application because the user will typically view the application from a distance of six to eight feet on a standard television set. A spread out layout would make it clearer and easier to read.

A good rule of thumb is to allow 50% more space than is required for an English text string for translations into other languages. In general, the actual additional space required will be more on the order of 30%, but stick with the 50% figure. You're better off having a little extra space than needing to redesign your layout after the translations come back and you have the one exception to the 30% general figure.

If the application does end up with dynamic text strings and formatting routines rather than different program versions for each language, the structure of the *locale.library* be used as a model. The text string files should be stored in a subdirectory of the directory containing the application; the subdirectory name should be based on the language it contains; and the file name should clearly indicate which application it belongs to.

In either case, you should keep text separated from your code! If you are going to include on-screen text in your application, make sure you keep those strings out of the body of your code. Searching through thousands of lines of code to find error messages, prompts, etc., can be a long and painful experience. A little advanced planning when analyzing your application can help enormously when translating the text to other languages.

If possible, keep all text strings in a separate file. That file may then be sent to a translation agency for localization. This step avoids the necessity of sending your entire source code to a third party—an understandably painful process for most developers.

Doing entirely without text will naturally make an application much easier to localize. Liberal use of easy to understand symbols will provide an international feel without a lot of translation. However, symbols must be checked with native speakers of any language considered for localization. Cultural differences may make an image that seems clear to a native of one country mean something completely different or even insulting to a native of another country.

As previously mentioned, there is more to think about in localizing an application than just text strings and symbols. The format for numeric output varies from country to country as does the format used for monetary values and the format for dates. While not as important as text string translation, correctly localizing these formats goes a long way toward getting your product accepted in foreign markets.

Another localization issue is the keymap selection for the optional CDTV keyboard. Unfortunately, an application can only make a reasonable guess as to what keymap the user requires based on the language preference. In some cases, the obvious choice will be incorrect. In that case, it would be nice to offer the user a method of selecting a different keymap in a heavily text based application. The Commodore function *CDSetMap* will read the CDTV Preferences for the language setting and make a keymap decision based on that selection. Its current language/keymap correspondence table is:

Language	Keymap
American English	USA
English	GB
German	D
French	F
Spanish	E
Italian	I
Portuguese	PORT
Danish	DK
Dutch	USA
Norwegian	N
Finnish	S
Swedish	S
Japanese	USA
Chinese	USA
Korean	USA
Arabic	USA
Greek	USA
Hebrew	USA
OTHER	USA

As you can see, any country that does not currently have a keyboard specific to that country receives the USA keymap as a default.

Any keymaps other than the default ROM USA keymap are going to have to be present on your application disc. Unlike the normal *SetMap*, the *CDSetMap* command uses keymaps in the current directory, as well as in the *DEVS:keymaps* directory. If the *CDSetMap* command is used to set the current keymap, you can leave all the keymaps in the root directory. The complete documentation for the *CDSetMap* command appears at the end of this article.

Include Sampled Sound

One of the best ways to create interesting, localized multimedia titles is to use voice narration. With over 600 Mbytes of space available on a CD, you have room for up to twenty-eight hours of digitized audio (at a reasonable sampling rate). If you have one hour of a digitized voice in your original code, you can easily sample the same sentences in other languages, and play those samples back in your code. See the article "Producing High Quality Digitized Narrative Output" for hints on how to digitize text in a foreign tongue.

While the Amiga narrator/translator is limited to American English, CDTV is not. It is possible for an application to fully support speech in as many languages as the application has room for on the disc (and in as many languages as reasonable translations and native speakers are available). Multiple translations of audio tracks naturally adds to the time and cost required to produce a CDTV application.

The *narrator.device* is an unusual feature of the Amiga, letting you synthesize the human voice directly from an ASCII text file. However, the quality of the synthesized voice does not approach the quality of sampled audio. Furthermore, the *narrator.device* and the *translator.library* do not

support languages other than English. Attempting to have the *narrator.device* pronounce German or Italian text can be a humorous, if not rewarding, experience.

Localization is one of the keys to creating an application that can be successful in today's international market. Time spent on localization can be rewarded tenfold through increased sales and worldwide acceptance of the localized application.

NAME

CDSetMap—Set the global default keymap based on the Preferences language setting.

SYNOPSIS

CDSetMap [keymap name] [DEFAULT/K]

FUNCTION

The **CDSetMap** command is used to set the global default keymap for the CDTV system based on the language preferences stored in the non-volatile RAM previously set by the user.

Normally executed without any argument in the startup-sequence, the command will read the CDTV Preferences, and make a keymap decision according to the current language setting. As there is not a one-to-one correspondence between languages available and keyboards available, a semi-reasonable mapping is used. This mapping is as follows:

Language	Keymap
American English	USA
English	GB
German	D
French	F
Spanish	E
Italian	I
Portuguese	PORT
Danish	DK
Dutch	USA
Norwegian	N
Finnish	S
Swedish	S
Japanese	USA
Chinese	USA
Korean	USA
Arabic	USA
Greek	USA
Hebrew	USA
OTHER	USA

This default mapping of unavailable keymaps to the USA keymap may be overridden by using the **DEFAULT** keyword, and specifying a different keymap to use as the default.

The **CDSetMap** command may also be given a keymap argument. In that case, the keymap indicated is set, no matter what the current Preferences setting is. The Preferences setting stored in NVR itself is unaffected.

NOTES

The **CDSetMap** command looks for keymaps in the current directory first, then looks in the `devs:keymaps` directory. If the indicated keymap is unavailable, the global keymap setting will remain unchanged.

CDTV File System

The CDTV filesystem (henceforth referred to as CDFS) is an AmigaDOS compatible file handler that interprets the ISO-9660 CD-ROM interchange format and presents it to the user and applications as a standard AmigaDOS volume. CDFS also provides for booting from a CD-ROM, and provides several boot-time configuration options.

ISO-9660 Overview

ISO-9660 is the standard established by the International Standards Organization (ISO) to provide a "common ground" that CD-ROM vendors and users can use to maximize disc portability between systems. ISO-9660 is based substantially upon the "High Sierra" format, established by CD-ROM developers in the mid-1980s. ISO replaces the High Sierra format.

ISO-9660 establishes how files and directories are laid out on the CD-ROM sectors, the format of directory structures, how files and directories may be named, the structures describing the entire volume and where to look for them, and many other details. An ISO-9660 filesystem is contained within a single CD "track." ISO does not mandate that the disc contain only CD-ROM data. CD digital audio may also be stored in separate CD tracks.

The entire ISO volume is described by a Primary Volume Descriptor (PVD) located near the beginning of the track. The PVD contains the name of the disc, the name of the publisher, the name of the manufacturer, the computer system and application for which the disc is intended, a copyright notice, the disc's sequence number (for discs that are members of a multiple-volume set), the total number of discs in the volume set, the size of the entire volume, a set of dates describing when the volume was created, when the data on the disc becomes valid, and when the data becomes obsolete. It also describes where the root directory structure may be found.

ISO also provides for optional Supplementary Volume Descriptors, Boot Records, and Partition Descriptors. Supplementary Volume Descriptors can be used to define additional information about a volume, specify a subset of the directory hierarchy declared in the PVD, or even a completely different directory hierarchy. Boot Records usually contain bootstrapping code or data, and also identify the system for which the boot code/data is intended. Partition Descriptors simply declare a portion of the volume space whose contents are otherwise undefined. Partition Descriptors can be used to reserve space in the ISO volume for raw data intended to be read directly off the CD.

ISO describes a hierarchical directory structure. Unlike AmigaDOS, directories are stored as "files" on the disc; the entire contents of the directory can often be loaded in a single read. The directory nesting is not permitted to exceed eight levels, and a full pathname must not exceed 255 characters. ISO further mandates that the entries in the directory be sorted in ascending ASCII order.

Files under ISO are untyped, may contain anything, and may be up to four gigabytes in size. Files under ISO can be recorded in sequential sectors, or may be recorded in an "interleaved" fashion. Interleaved files are written such that M successive sectors contain file data, then N sectors are skipped, then another M sectors of data, then N sectors skipped, etc. This permits

application writers to interleave files together such that, when the files are read simultaneously, a single contiguous read of the CD-ROM results, which increases performance. Optionally, files may also include an Extended Attribute Record (XAR), which establishes access permissions, and also helps define the contents of the file.

ISO also establishes what are called "associated files." These files are "shadows" of ordinary files. They have the same name, but their contents are completely different.

ISO designates a set of ASCII characters which may be legally used for file and directory names, and other fields in ISO structures. Two sets are defined. The set called "a-characters" consists of the following:

Range	Characters
0x20–0x22	!"
0x25–0x3F	%&'()*+,-./0123456789:;<=>?
0x41–0x5A	ABCDEFGHIJKLMNOPQRSTUVWXYZ
0x5F	–

The set known as "d-characters" is a subset of a-characters, and is limited to the following:

Range	Characters
0x30–0x39	0123456789
0x41–0x5A	ABCDEFGHIJKLMNOPQRSTUVWXYZ
0x5F	–

By the above, lowercase characters are excluded from the standard. In actual practice, however, lowercase is used frequently in ISO structures, though its use is not standardized.

File and directory names are restricted to the set of d-characters. In addition, filenames are required by the standard to have one and only one dot ('.') in them, which separates the filename from the filename extension. Filenames are also required to have a version number following them, separated by a semicolon(';'). The sum of the lengths of the filename and extension may not exceed 30 characters. Thus, examples of valid filenames as they would appear on an ISO volume might include:

```
COMMAND.COM;1
UTILITIES.INFO;3992
IFFPARSE.LIBRARY;37
.LOGIN;2
AMIGALOGO.;1
VERY_LONG_FILE_NAME_INDEED.BAK;1
```

Because the seek times on CD-ROM drives are so slow, ISO-9660 establishes the presence of a Path Table. This table contains the name and location of every directory on the volume, and is meant to be loaded by the host when the disc is inserted. By searching the Path Table, the host is able to arrive at any directory on the disc with a single seek.

As a final note, your CDTV applications do not need to know about ISO-9660 in order to operate, since CDFS takes care of all the mish-mash for you. If they run off a write-protected floppy or write-protected hard disk, they will run off a CD just fine.

What CDFS Does

CDFS resides in the extended ROM kernel within every CDTV unit. It represents an ISO-9660 filesystem as an AmigaDOS volume to users and applications. As far as programs are concerned, it's just a very big write-protected floppy disk. CDFS conforms to a Level 1 Implementation, and accepts disks mastered at Level 2 Interchange.

CDFS permits the use of lower-case, punctuation, and international characters in file and directory names. CDFS does not require a dot ('.') to be present in file names, but forbids the use of the semicolon(';'). In short, any sequence of characters that represents a valid AmigaDOS file name will be recognized by CDFS. (Note that, if you have a disc mastered using these relaxed rules, the portability of your disc to other platforms is reduced.)

CDFS also lets you set various configuration options and cache sizes at boot time, and also permits the developer to specify alternate boot methods.

What CDFS Doesn't Do

Due to AmigaDOS compatibility, short development time, and testing restrictions, there are a few ISO features not supported by CDFS 1.0.

The current version of CDFS does not contain support for interleaved files. Attempts to read interleaved files currently yield garbage data.

Associated files, if present, are reported in directory listings. The visible effect will be a duplicate entry in the listing. However, attempts to access the relevant file name will yield only the associated file; the non-associated file is inaccessible via a direct `Open()`, but can be accessed with `ExNext()`.

Files with Extended Attribute Records (XARs) are not directly supported. The XAR will appear as file data and the file length will appear shortened. To properly access a file with an XAR requires a call to the device driver.

The version number embedded in the file name is unavailable to AmigaDOS; filenames are internally truncated at the semicolon. Additionally, files sharing the same name but different version numbers will appear as multiple entries with the same name in a directory listing. However, only the first such file (the one with the highest version number) is directly accessible via an `Open()`. The others must be accessed with `ExNext()`.

Supplementary Volume Descriptors and Partition Descriptors are currently ignored.

Enhancements for all the above are being investigated for a future version of CDFS.

Application Programming

CDFS requires no unusual programming. All the DOS calls work as expected (`Open()`, `Close()`, `Read()`, `Info()`, `Examine()`, etc.), with the minor exception that all attempts to modify the disc return an error, reporting the disc to be write-protected.

Packet-driven I/O through CDFS also operates normally. CDFS also supports an extra packet, `ACTION_DIRECT_READ` for better performance with large files. See the section on direct reading for more information.

File And Directory Layout

The major time sink when reading a CD-ROM is seeking. Seeking can easily consume 80% of the time required for data transfers. Obviously, your application will want to minimize seek times. One good way to do this is to take advantage of CDFS's read-ahead and directory caches and try to lay the files and directories out on the disc such that they can be read with minimum seeking.

Although ISO mandates that the entries in a directory be sorted in ASCII order, you are free to place the actual contents of a file anywhere on the disc. The same is true of the directories. Thus, you will probably want to lay out the files on your disc in the order in which they're accessed by your application.

As an example, if your application were, say, Workbench, you would want to group all the *.info* files together so that they could be scooped up by a single large read, enabling the icons to appear on screen very quickly.

By the same token, if you have a directory or set of directories that you access frequently or in a particular order, you may wish to group those directories together on the disc.

Direct Reading

CDTV has the ability to perform direct DMA from the disc into your application's buffers. However, the current version of CDTV's DMA controller has a bug such that the last few bytes of a transfer might not actually happen.

The workaround for this is to allocate buffers slightly larger than needed, and request slightly more bytes than needed. Thus, the desired number of bytes is received; any missing bytes are part of the padding and are ignored. CDFS does this internally by allocating slightly larger buffers than it needs, DMA'ing into them, and then using the CPU to copy the desired number of bytes into the client's buffer.

However, if you write your application to allocate slightly larger buffers, you can enable CDFS's direct reading feature, which will cause CDFS to DMA the data directly from the disc to your buffers.

Once you have enabled direct reads, you need only perform the padded allocations. Unlike `CMD_READ` with `cdtv.device`, you *do not* need to explicitly request the extra bytes; CDFS does this for you. The amount by which to pad your allocations is `READ_PAD_BYTES`, and is defined in `<devices/cdtv.h>`. An example of reading into a buffer after direct reads are enabled might appear as follows:

```
UBYTE  buffer[BUFSIZ + READ_PAD_BYTES];
Read (file, buffer, BUFSIZ);
```

There are a few other rules that apply to direct reads. In order for direct reads to work, the file must be positioned on an even byte boundary, the address of the destination buffer must be word-aligned, and the size of the transfer must be an even number of bytes. If all these conditions are met, your buffer will be filled using direct reads. Otherwise, the default indirect method will be used.

There are two ways to enable direct reads. One is to enable direct reads globally by installing the direct reading boot option in the volume's PVD. This is discussed in more detail in "Boot Options" below.

The other way is to send a packet to CDFS asking it to enable direct reads for a particular file. The packet is **ACTION_DIRECT_READ**, and has the following semantics.

Type	LONG	ACTION_DIRECT_READ (1900)
Arg1	CPTR	Filehandle structure
Arg2	LONG	Boolean
Res1	LONG	DOS_TRUE

Arg1 is a C-style pointer (not a BPTR) to a Filehandle structure obtained by opening a file on the CD volume. If Arg2 is TRUE, then direct reads for the file are enabled; if FALSE, direct reads are disabled.

Make Sure It's CDFS. **ACTION_DIRECT_READ** is currently unique to CDFS. Other file handlers will generate an error result for this packet. It is wise not to invoke this packet unless you are certain you are talking to CDFS.

Boot Time Options

When CDFS boots a CD-ROM, it inspects the PVD to see if any boot-time options have been placed in the application use area of the PVD. These options and their descriptions are:

Read-Ahead Cache Size

CDFS allocates a read-ahead cache to improve performance. The size of this cache is specified in CD sectors (2K bytes each). The default size is 8 blocks.

Directory Cache Size

CDFS allocates a cache in which it saves the contents of directories it has read. This boosts performance if certain directories are read frequently. The size of this cache is specified in CD sectors (2K each). The default size is 16 blocks.

Filelock Pool Size

To reduce memory fragmentation, CDFS pre-allocates a pool of Filelock structures which will be consumed as applications procure locks on files. The size of the pool is specified as the number of Filelock structures to pre-allocate. The default value is 40.

Filehandle Pool Size

As with Filelocks, CDFS pre-allocates a pool of Filehandle structures. The size of the pool is specified as the number of Filehandles to pre-allocate. The default value is 16.

Direct Read

Enables direct reads for all file I/O. See the section on direct reading in this document for more details.

Fast Directory Searches

ISO-9660 mandates that entries in a directory be sorted in ascending ASCII order. However, since it only allows upper-case file names, it doesn't say how to sort filenames that contain lower-case. Do you sort case-sensitive or case-insensitive?

The Fast Directory Search option will stop searching a directory when it passes the point where the filename could be. However, for it to work, the directory must have been sorted

case-insensitively. If your disc is so mastered, you may enable this option and enjoy improved performance on directory searches.

Retry Count

This specifies the number of times to retry a read operation in the event of an error. The default value is 32.

As mentioned earlier, these options are stored in the **Application Use** field of the **PVD** structure (byte offset 885). Each option is specified by a header consisting of a 16-bit identifier followed by a 16-bit length. The identifier uniquely identifies the option. The length specifies the number of data bytes following the header. This length must be even. A length of zero is valid. If there are additional options, they are written immediately after the last data byte. An identifier of zero terminates the list of options. Options start at byte offset 1 in the **Application Use** field, which achieves word alignment.

Boot options are created and installed by either the ISO image mastering software, or by *SetCDFs*, a tool available from Commodore which installs and removes options from an existing ISO image file.

CDTV Device

Introduction

This document describes the operation of *cdtv.device*, which is the core of operation for the CD-ROM drive hardware. It is part of the extended ROM Kernel within each CDTV unit. Among other things, *cdtv.device* is responsible for:

- Reading CD-ROM data
- Playing CD-DA (Digital Audio)
- Tracking laser position (Subcode Q)
- Synchronizing audio with other events
- Fetching CD Table of Contents (TOC)
- Attenuating CD audio outputs
- Obtaining drive status
- Determining error conditions
- Disabling CDTV front panel controls
- Changing genlock modes

The CDTV device is the lowest level access point for the CD-ROM drive. Direct hardware access is forbidden. Violators will be *mercilessly* punished.

All device commands are performed through I/O requests sent directly to the CDTV device. Nearly all of these commands can be executed either synchronously or asynchronously depending on application requirements. The device also supports a two channel command request queue to permit the operation of simultaneous asynchronous commands. This allows, for instance, a laser position query to occur during a play command.

Before You Start

CDTV is built around the Amiga computer. As such, you are required follow all the programming standards and practices of the Amiga.

- Be sure you are familiar with the *Amiga ROM Kernel Reference Manuals*.
- Read this document completely.
- Read the *Autodocs* (programmer's reference) which describe the device commands in precise detail.
- Read the C and/or assembly include files, which also contain additional valuable information.

- Use the Commodore-supplied debugging tools where possible.
- Avoid empirical programming (just because it works, doesn't mean it's right).
- Observe *all* programming standards.

In particular, you should *not* assume the current CDTV hardware will remain unchanged! For example: the OS ROMs will soon be upgraded to v2.04; the CD-ROM drive will change; hardware addresses will change; perhaps the processor will improve; memory may expand; etc. If you follow the rules, such changes will not affect you. If you decide to break the rules, great misfortune is certain to befall you.

If you are new to Amiga programming, carefully study of the example programs in the *Amiga ROM Kernel Reference Manuals*. Also, the *Fred Fish Collection* (available on CD-ROM) contains hundreds of programs with working example source.

With proper care your title will function not just on CDTV, but also on the A570, properly-equipped Amiga systems, and future CDTV systems, thus minimizing your number of SKUs and after-sale support, and maximizing your unit sales and profits.

This document assumes that you are already familiar with the mechanics of Exec-level device I/O and Amiga programming in general.

The Compact Disc

Compact Discs (CDs) are a read-only optical medium capable of storing several hundred megabytes on a single disc. CDs have enjoyed tremendous popularity as a distribution medium for music. CD-ROM is an extension of the medium, putting computer data on a disc instead of digital audio.

CDs are laid out as a continuous spiral track, from the inner to outer edges of the physical disc. A CD can hold a maximum of roughly 74 minutes of music, or about 660 megabytes of data. (Some CDs hold more, but many CD mastering houses prefer not to push these limits.) CD-ROMs may freely intermix audio and data tracks, but most systems expect a ROM track to be the first track on the disc.

Data on CDs and CD-ROMs are divided into "tracks" and "frames." A track is analogous to a music track. CDs start numbering their tracks from 1; there is no track zero. A CD can have a maximum of 99 tracks. An ISO-9660 filesystem (the standard format for CD-ROM data) is contained in a single track, and is usually the first track on the disc. Tracks can also be further subdivided by including index positions. However, the index feature of CDs is rarely used.

A "frame" is CD parlance for a sector. Sectors on a CD are 2,048 bytes in size. All CD players read sectors (frames) off the disc at 75 frames per second. At 2K per sector, this makes the maximum transfer rate 153,600 bytes per second.

The first 150 or so sectors (it varies) are devoted to the Table Of Contents (TOC). The TOC describes how many tracks are on the disc, where each track begins, and what type of track it is (audio or data). It also specifies the length of the entire disc.

Embedded within the CD datastream are eight one-bit subchannels, named P, Q, R, S, T, U, V, and W. Subchannel P is a simple annunciator bit indicating the start of a music track; it is almost never

used today. Subchannels R through W are used for CD+G and CD+MIDI data. These subchannels are not available to the programmer.

Subchannel Q, on the other hand, is available. Each frame, the bits of Subchannel Q are assembled to form a complete *SubQ* packet. Most often, a *SubQ* packet describes the current track and index, the type of the current track (audio or data), current position within the track, and the current position from the beginning of the disc. More rarely, *SubQ* packets can contain Universal Product Codes (UPC) which uniquely identify the disc, and International Standard Recording Codes (ISRC), which uniquely identify songs on a disc.

Finding Your Way Around A Disc

The CDTV device uses two numbering systems to specify locations on a disc: LSN format and MSF format. LSN stands for Logical Sector Number. Logical Sector Numbers begin at zero and increment sequentially, one per frame, to the end of the disc.

MSF format stands for Minutes-Seconds-Frames. This system specifies a number of minutes, seconds, and CD frames since the beginning of the disc. This format is most frequently used when playing CD audio. However, the math is more complicated using MSF values rather than LSN values. MSF numbers written out usually appear as "05:09.37", which means 5 minutes, 9 seconds, 37 frames.

MSF numbers are passed to the device driver as a packed longword. The minutes value is placed in bits 16–23, the seconds value in bits 8–15, and the frames value in bits 0–7. Bits 24–32 must remain zero. As an example, MSF 05:09.37 would be represented as 0x00050925. You may care to use the `TOMSF0` macro in `<devices/cdtv.h>` to generate MSF values.

Both numbering systems start at zero; there are no negative addresses. However, LSN zero and MSF zero refer to different positions on the disc. LSN zero is equivalent to MSF 00:02.00. MSF zero refers to the very beginning of the disc. However, this is where the table of contents is located. Thus, the very first usable frame is the first frame of the first track. The location of this frame can be found by retrieving the TOC and inspecting the starting position of the first track. This usually comes out to be MSF 00:02.00, but different manufacturers may put it in different places, particularly if it's an audio disc. Note that it is not a good idea to try to position the laser before the first valid frame.

Device Access

cdtv.device is just like any other Amiga Exec-level device. Commands are sent using the `IOStdReq` structure (defined in `<exec/io.h>`). The I/O structure and data buffers may reside in any type of memory (Chip or Fast). The structure must be properly initialized. Nearly all commands will require a message reply port.

Transactions with *cdtv.device* are initiated through `OpenDevice()`. A typical opening of the device might appear as follows (the function `err0` is not part of the system; it's up to you):

```
struct MsgPort *reply;
struct IOStdReq *cdio;

/* Create reply port. */
if (!(reply = CreatePort (NULL, NULL)))
```

```

    err ("Can't allocate reply port.");

/* Create I/O structure. */
if (!(cdio = CreateStdIO (reply)))
{
    DeletePort (reply);
    err ("Can't allocate CD I/O structure.");
}

/* Open device. */
if (OpenDevice ("cdtv.device", 0, cdio, 0))
{
    DeleteStdIO (cdio);      /* There are more elegant ways */
    DeletePort (reply);     /* to do this. */
    err ("OpenDevice failed.");
}

```

The **unit** and **flags** parameters to **OpenDevice()** must be set to zero for future compatibility. You may open the device as many times as you wish. However, each call to **OpenDevice()** must eventually have a matching call to **CloseDevice()**.

For The Experts. The **io_Device** and **io_Unit** fields may safely be cloned to create multiple I/O requests.

Performing Commands

CDTV device commands are performed in exactly the same fashion as other Exec devices. The **DoIO()** function will perform synchronous commands; it will not return until the command has completed. The **SendIO()** function will perform asynchronous commands; it will initiate the command and return control to your program. **CheckIO()** will check to see if an asynchronous request has completed. **WaitIO()** will wait for a request to finish. Refer to the *Amiga ROM Kernel Reference Manual: Libraries and Devices* for a full description.

Before performing a command, the request structure must be set up with the various parameters of the command. The **io_Command** field must be filled in with the desired command code as found in *<devices/cdtv.h>*. The values of the **io_Length**, **io_Offset** and **io_Data** fields vary from command to command. Unused fields must be set to zero for future compatibility.

BeginIO()—For Experts Only

The **BeginIO()** direct interface is functional in the CDTV device. However, there is very little reason for using it. Nearly all commands are queued internally because they require communication with the hardware that is always asynchronous and interrupt driven.

It may be possible to shave a microsecond or two with this approach, but you must be certain to handle the **QUICKIO** flag in the proper manner and you will need to **Disable()** under some conditions. Further, these conditions will change in the future as the hardware improves. **DoIO()**, et al, will always handle these cases correctly.

It is strongly advised *not* to use **BeginIO()** with *cdtv.device* in all cases. This argument also applies to **PerformIO()** for those of you who are aware of such magic.

Command Overview

Below is a table of the available commands in *cdtv.device*. The table consists of the command name, its numeric value, its type, and a brief description. Complete descriptions and parameters are detailed in the *cdtv.device Autodocs*.

The first twenty-three commands are standard Amiga disk commands. The CD-relevant commands begin at command 32. Command numbers not listed will return an error. Note that this is not a SCSI device, so the SCSI direct command (28) is not valid.

The "Type" column defines the types of conditions that apply to the command. They are:

- Q** The command is Queued to be processed by the CDTV device driver task. Such commands cannot be done from interrupts.
- S** Only a Single command of this type can be executing at the same time. If more than one S type command is requested, each will wait its turn before proceeding. Non-S commands can be executed during an S command.
- D** The command requires a Disc be present.

Command	Num	Type	Description
CDTV_RESET	1		Reset the device
CDTV_READ	2	QSD	Read bytes from CD-ROM
CDTV_WRITE	3		Error
CDTV_UPDATE	4		Nop
CDTV_CLEAR	5		Nop
CDTV_STOP	6		Stop
CDTV_START	7		Start
CDTV_FLUSH	8		Nop
CDTV_MOTOR	9		Turn CD motor on/off
CDTV_SEEK	10	D	Seek to a sector
CDTV_FORMAT	11		Error
CDTV_REMOVE	12		Error
CDTV_CHANGENUM	13		Return disk change count
CDTV_CHANGE STATE	14		Return disk change status
CDTV_PROTSTATUS	15	D	Always returns write protected
CDTV_GETDRIVETYPE	18		Always a CD type
CDTV_GETNUMTRACKS	19	D	Nop
CDTV_ADDCHANGEINT	20		Add disk change interrupt
CDTV_REMCHANGEINT	21		Remove disk change interrupt
CDTV_GETGEOMETRY	22	D	Nothing for now
CDTV_EJECT	23		Error (currently)
CDTV_DIRECT	32	Q	Direct CD commands
CDTV_STATUS	33		Direct CD status
CDTV_QUICKSTATUS	34		Return status quickly

CDTV_INFO	35	Q D	Return CD info
CDTV_ERRORINFO	36	Q	Return error code
CDTV_ISROM	37	Q D	Determine if CD is ROM or DA
CDTV_OPTIONS	38	Q	Set options
CDTV_FRONT PANEL	39	Q	Turn front panel on/off
CDTV_FRAMECALL	40		Setup frame callback function
CDTV_FRAMECOUNT	41		Return frame counter
CDTV_READXL	42	QSD	Perform transfer list read
CDTV_PLAYTRACK	43	QSD	Play track/index
CDTV_PLAYLSN	44	QSD	Play logical sector number (LSN)
CDTV_PLAYMSF	45	QSD	Play minute:second.frame (MSF)
CDTV_PLAYSEGSLSN	46	QSD	Play segment list LSN
CDTV_PLAYSEGMSF	47	QSD	Play segment list MSF
CDTV_TOCLSN	48	Q D	Table of Contents LSN
CDTV_TOCMSF	49	Q D	Table of Contents MSF
CDTV_SUBQLSN	50	Q D	Subcode Q (position) LSN
CDTV_SUBQMSF	51	Q D	Subcode Q (position) MSF
CDTV_PAUSE	52	Q D	Pause during play
CDTV_STOPPLAY	53	Q	Stop a play
CDTV_POKESEGSLSN	54	D	Poke segment list LSN
CDTV_POKESEGMSF	55	D	Poke segment list MSF
CDTV_MUTE	56		Get/Set DAC Attenuator
CDTV_FADE	57	Q	Fade Attenuator up/down
CDTV_POKEPLAYLSN	58	D	Poke an active play, LSN
CDTV_POKEPLAYMS	59	D	Poke an active play, MSF
CDTV_GENLOCK	60		Set genlock mode

Data Commands

There are three commands to assist in reading data from CD-ROM:

CDTV_READ

Reads bytes from CD-ROM sectors. Because the CD-ROM is a word-oriented device, all transfers must be word-aligned and must be an even number of bytes. Unlike many other Amiga disc device drivers, the transfer does not need to be sector aligned.

CDTV_READXL

This command performs a "scatter read" of contiguous data from the CD-ROM into a list of buffers. Its operation is detailed in a separate article.

CDTV_SEEK

Positions the laser as close as possible to the requested position. CDTV_READ and CDTV_READXL automatically seek to a specified location. However, because of the high seek times on CD-ROM drives, it may be valuable to pre-seek to a position, thus reducing the settling time when it comes time to actually reading the data. By performing a seek at the right time, you can cut up to half a second off the next CDTV_READ or CDTV_READXL command.

The CD-ROM drive hardware prohibits all attempts to read CD digital audio data. This restriction is required by Commodore's license with the compact disc patent holder, and is ostensibly to prevent the digital copying of copyrighted music.

Watch Out For This

There is currently a bug in the hardware which *might* cause the last few bytes of a read to not be transferred.

The workaround is to request extra bytes, and to pad your memory buffers and disc data accordingly. In this way, you will receive the desired amount of data; the missing bytes, if any, are part of the padding, and are ignored.

The amount by which to pad your buffers and disc data is `READ_PAD_BYTES`, and is defined in `<devices/cdtv.h>`. This bug will be fixed in the future.

Reading from the CD-ROM might take the following form:

```
UWORD  buffer[BUFSIZ + READ_PAD_BYTES/2];

cdio->io_Command = CDTV_READ;
cdio->io_Offset  = disc_location;      /* In bytes. */
cdio->io_Length  = sizeof (buffer);    /* In bytes. */
cdio->io_Data    = buffer;
DoIO (cdio);
```

Status Commands

Several facilities exist to interrogate the drive about its current status:

CDTV_CHANGENUM

Reports the number of times a disc has been inserted or removed from the drive.

CDTV_CHANGESTATE

Reports whether or not a disc is currently in the drive.

CDTV_ISROM

Reports whether or not the disc currently in the drive is a CD-ROM. This is useful for determining whether or not you can perform data operations on the disc.

CDTV_FRAMECOUNT

Reports the number of CD frames that have transpired since the disc was inserted. This can be used as a timebase for event synchronization.

CDTV_QUICKSTATUS

Quickly reports the current state of the drive, if available. Among other things, it tells you if there's a disc in the drive, if the disc is spinning, and if audio is playing.

CDTV_SUBQ{LSN,MSF}

Reports the current state of the drive, and position of the laser over the disc. This command is most frequently used during CD audio operations.

For the commands `CDTV_CHANGENUM`, `CDTV_CHANGESTATE`, `CDTV_FRAMECOUNT`, `CDTV_QUICKSTATUS` and `CDTV_ISROM`, the requested information is returned in the `io_Actual` field of the I/O request. For example, the `CDTV_ISROM` command might be used as follows:


```

BOOL    CDROM_present;
LONG    error;

cdio->io_Command = CDTV_ISROM;
cdio->io_Data    = NULL;
cdio->io_Length  = 0;
cdio->io_Offset  = 0;
if (!(error = DoIO (cdio)))
    CDROM_present = cdio->io_Actual != 0;

```

CDTV_QUICKSTATUS

CDTV_QUICKSTATUS is used to interrogate the drive's current state. The report from CDTV_QUICKSTATUS is a set of status bits. The bits and their meanings are as follows:

QSF_READY	Drive is ready.
QSF_AUDIO	CD digital audio is being played.
QSF_DONE	Last hardware command finished.
QSF_ERROR	Error in last hardware command.
QSF_SPIN	The disc is spinning.
QSF_DISK	A disc is present in the drive.
QSF_INFERR	Error related to positioning.

Subject To Change It is not a good practice to depend on the values of QSF_READY, QSF_DONE, QSF_ERROR or QSF_INFERR. Their meanings are not well defined and will change in the future as the hardware improves.

CDTV_QUICKSTATUS returns the current state in the io_Actual field of the I/O request. The command might be used as follows:

```

ULONG    status_bits;
LONG    error;
BOOL    disc_present, disc_spinning, audio_playing;

cdio->io_Command = CDTV_QUICKSTATUS;
cdio->io_Data    = NULL;
cdio->io_Length  = 0;
cdio->io_Offset  = 0;
if (!(error = DoIO (cdio)))
{
    status_bits = cdio->io_Actual;
    disc_present = (status_bits & QSF_DISK) != 0;
    disc_spinning = (status_bits & QSF_SPIN) != 0;
    audio_playing = (status_bits & QSF_AUDIO) != 0;
}

```

CDTV_SUBQ{LSN,MSF}

This command is used to request a SubQ packet from the drive. As discussed earlier, the SubQ packet will tell you the current position of the laser on the disc. Additionally, it will also tell you the current state of the drive. This command is principally used with CD audio applications. CDTV's built-in audio control panel makes extensive use of SubQ information to display the elapsed playing time and other time modes.

The information is placed into a CDSUBQ structure, which is defined in *<devices/cdtv.h>*, and is shown below:

```

struct CDSubQ
{
    UBYTE  Status;           /* Audio status */
    UBYTE  AddrCtrl;         /* SubQ info    */
    UBYTE  Track;           /* Track number */
    UBYTE  Index;           /* Index number */
    CDPOS  DiskPosition;    /* Position from start of disk */
    CDPOS  TrackPosition;   /* Position from start of track */
    UBYTE  ValidUPC;        /* Flag for product identifier */
    UBYTE  pad[3];          /* undefined    */
};

```

The fields in the structure have the following meanings:

Status

This field contains the current status of the drive, and is vaguely similar to the report generated by CDTV_QUICKSTATUS. The field is numeric, and can have the following values (any other value is invalid):

SQSTAT_NOTVALID	This SubQ report is not valid.
SQSTAT_PLAYING	The drive is currently playing CD audio.
SQSTAT_PAUSED	The drive is currently paused.
SQSTAT_DONE	The most recent play command completed successfully.
SQSTAT_ERROR	The most recent play command generated an error.
SQSTAT_NOSTAT	No status available; this report is invalid.

This field must be sanity-checked. See the note below for more details.

AddrCtrl

This field describes the nature of the current track, as well as the type of SubQ report. The field is split into two four-bit fields. The lower four bits are status bits. Their meanings are:

ADRCTLF_PREEMPH	This track has pre-emphasis of 50/15 μ S.
ADRCTLF_COPY	This track may be digitally copied. In our experience with hundreds of discs, we have never observed this bit to be set :-). (In no case is digital audio data readable by the host.)
ADRCTLF_DATA	This track contains CD-ROM data (as opposed to CD audio).
ADRCTLF_4CHAN	This track contains 4-channel sound.

The upper four bits of the field are numeric. The possible values are:

ADRCTL_NOMODE	No mode information was available; this report should be considered invalid.
ADRCTL_POSITION	This report contains position information.
ADRCTL_MEDIACAT	This report contains a UPC code.
ADRCTL_ISRC	This report contains an ISRC number.

This entire field must be sanity-checked. See the note below for details.

Track

Contains the current track number.

Index

Contains the current index within the track.

DiskPosition

Contains the current position of the laser relative to the beginning of the disc.

TrackPosition

Contains the current position of the laser relative to the beginning of the current track.

ValidUPC

This field contains status bits. They are:

SQUPCB_VALID	A UPC or ISRC code was detected.
SQUPCB_ISRC	When set, ISRC detected; clear, UPC detected.

The address of the **CDSUBQ** structure is placed in the **io_Data** field of the I/O request. The **io_Length** and **io_Offset** fields should be set to zero for future compatibility.

You should perform sanity checks on the **SubQ** data to be sure that it is a valid report. In particular, inspect the **Status** and **AddrCtrl** fields to be sure they indicate the report is valid. Also, watch for nonsense combinations of the lower four **AddrCtrl** bits (e.g., a report where both **ADRCTL_DATA** and **ADRCTL_PREEMPH** are set is clearly meaningless).

Here is an example of the command in use:

```
struct CDSUBQ    subq;
LONG            error;

cdio->io_Command = CDTV_SUBQMSF;          /* Request MSF format. */
cdio->io_Data    = &subq;
cdio->io_Length  = 0;
cdio->io_Offset  = 0;

if (!(error = DoIO (cdio)))
{
    /*
     * Test for valid position report. Discard all others.
     * This test passes only those reports that say the drive is
     * playing or paused, that the report is a position report,
     * and that the current track is not a data track.
     * (Note that this is the most rudimentary of sanity checks.
     * A proper one would be more thorough.)
     */
    if ((subq.Status != SQSTAT_PLAYING &&
         subq.Status != SQSTAT_PAUSED) ||
        subq.AddrCtrl & ADRCTL_MASK != ADRCTL_POSITION ||
        subq.AddrCtrl & ADRCTL_DATA)
    {
        return (BAD_REPORT);
    }
    printf ("Status:\t\t0x%02x\nValidUPC:\t0x%02x\n",
            subq.Status, subq.ValidUPC);
    printf ("Track, Index:\t%d, %d\n",
            subq.Track, subq.Index);
    printf ("DiskPosition:\t%02d:%02d.%02d MSF\n",
            subq.DiskPosition.MSF.Minute,
            subq.DiskPosition.MSF.Second,
            subq.DiskPosition.MSF.Frame);
    printf ("TrackPosition:\t%02d:%02d.%02d MSF\n",
            subq.TrackPosition.MSF.Minute,
            subq.TrackPosition.MSF.Second,
            subq.TrackPosition.MSF.Frame);
}
```

Audio Commands

There are many commands available to the programmer related to playing CD audio. Briefly, they are:

CDTV_PLAYTRACK

Play one or more complete music tracks.

CDTV_PLAY{LSN,MSF}

Play a segment of CD audio starting from a specific position, and ending at a specific position.

CDTV_PLAYSEGS{LSN,MSF}

Play a list of audio segments.

CDTV_POKEPLAY{LSN,MSF}

Alter a play in progress. Needed for "fast forward" type effects.

CDTV_POKESEG{LSN,MSF}

Jump to a particular audio segment node in an audio segment list.

CDTV_PAUSE

Pause or resume audio play.

CDTV_STOPPLAY

Stop a play operation that has been aborted.

CDTV_MUTE

Set CD audio volume level. Does not affect Amiga audio volume level.

CDTV_FADE

Fade CD audio volume level up or down over time. Does not affect Amiga audio volume level.

CDTV_TOC{LSN,MSF}

Retrieve the Table Of Contents for the disc.

Most commands that play CD audio come in LSN and MSF flavors, giving you the choice of specifying play positions and lengths using either LSN or MSF values. Commands using LSN and MSF values may, in many cases, be intermixed.

All audio play commands may be aborted using `AbortIO()`. However, the drive light will remain on, and the laser will continue to advance across the disc. This action is halted using the `CDTV_STOPPLAY` command, covered below.

The drive will abort all attempts to play non-audio tracks with an error. However, you should still take care to avoid accidentally playing CD-ROM data, since it is possible for a short burst of non-audio data to be sent to the DACs and out to the speakers. CD-ROM data sounds unbelievably bad. If you thought playing arbitrary data in RAM using AudioMaster or the *audio.device* sounded bad, wait until you hear it in 16-bit stereo.

CDTV_PLAYTRACK

This command provides the simplest form of CD audio control. It enables you to play whole tracks of audio, either singly or several at a time.

The number of the track at which you wish to begin playing audio is placed in the `io_Offset` field of the I/O request. The track at which you wish to stop is placed in `io_Length`. Optionally, you may also specify indicies within the starting and stopping tracks; they are placed in the upper 16 bits of the `io_Offset` and/or the `io_Length` fields.

Note that the stopping track does *not* specify the last track to be played, but the track at which the drive will stop. In other words, the moment the drive encounters the track/index specified in

io_Length, it will stop playing. If **io_Length** contains zero, the drive will stop playing at the next track.

The I/O request will return when the requested track(s) has completed playing. The command may be aborted with **AbortIO()**.

CDTV_PLAY{LSN,MSF}

For finer control of CD audio, there are the commands **CDTV_PLAYLSN** and **CDTV_PLAYMSF**. These commands let you specify an arbitrary absolute starting and stopping position on the disc.

The location from which to start playing audio is placed in the *io_Offset* field of the I/O request. This value is specified in either LSN or MSF format, depending on whether you're using **CDTV_PLAYLSN** or **CDTV_PLAYMSF**, respectively.

If you use **CDTV_PLAYLSN**, then the length of audio to play, in CD frames, is placed in **io_Length**. When the specified number of frames has been played, the drive will stop playing.

If you use **CDTV_PLAYMSF**, then the *stopping location*, in MSF format, is placed in **io_Length**. When the drive encounters this position, it will stop playing. Naturally, the stopping location must be greater than the starting location.

The I/O request will be returned when the requested audio segment has finished playing. Both commands may be aborted using **AbortIO()**.

CDTV_PLAYSEGS{LSN,MSF}

These commands let you play several CD audio segments in sequence. This command saves you from otherwise invoking **CDTV_PLAY{LSN,MSF}** in a loop.

The audio segments are described in a **CDAudioSeg** node structure, which is defined in *<devices/cdtv.h>*. The structure appears as follows:

```
struct CDAudioSeg
{
    struct MinNode Node; /* double linkage */
    CDPOS Start; /* starting position */
    CDPOS Stop; /* stopping position */
    void (*StartFunc)(); /* function to call on start */
    void (*StopFunc)(); /* function to call on stop */
};
```

The fields have the following meanings:

Node

An Exec **MinNode** used to link the **CDAudioSeg** structures together in a list.

Start

Starting location.

Stop

Stopping location (MSF) or play length (LSN).

StartFunc

Pointer to a call-back function to be called when this segment starts playing.

StopFunc

Pointer to a call-back function to be called when this segment has finished playing.

The **Start** and **Stop** fields have the same semantics as the **io_Offset** and **io_Length** fields, respectively, for the **CDTV_PLAY{LSN,MSF}** commands: **Start** specifies the starting location in either LSN or MSF format; **Stop** either specifies play length in CD frames (for **CDTV_PLAYSEGSLSN**), or the stopping location (for **CDTV_PLAYSEGMSF**).

StartFunc and **StopFunc** are optional pointers to functions which are called when the audio segment described by the node has begun/ended. This can enable you to synchronize your program to the audio sequence. See the section on call-backs for programming details.

To use the command, the address of the list header *or* the address of a **CDAudioSeg** node is placed in the **io_Data** field of the I/O request. If you pass the address of the list header, the driver will start playing audio segments from the beginning of the list. If you pass the address of a **CDAudioSeg** node, the driver will play segments starting with the passed node.

The command will be held until all nodes in the list are played. The command may be aborted using **AbortIO()**.

CDTV_POKEPLAY{LSN,MSF}

This command is the audio equivalent of a "seek"; it causes the laser to move to a new location and play CD audio from that point. This frees you from having to **AbortIO()** the current play command and issue a new one. It is particularly useful for "fast-forward" type effects.

The semantics of the **io_Offset** field and the **io_Length** field are identical to those for **CDTV_PLAY{LSN,MSF}**: **io_Offset** specifies the new starting location, **io_Length** specifies the new play length (for **CDTV_POKEPLAYLSN**) or the new stopping position (for **CDTV_POKEPLAYMSF**). You cannot intermix location address types with this command, i.e., you cannot use an LSN poke on an MSF play in progress, and vice versa.

Note that this command should be thought of as a modifier for **CDTV_PLAY{LSN,MSF}**. Thus, a **CDTV_PLAY{LSN,MSF}** command must be in effect when invoking this command (else you crash, go boom).

This command takes effect and returns immediately.

CDTV_POKESEG{LSN,MSF}

This command forces a "jump" to a particular node in an audio segment list. This enables you to skip around in a segment list, or to an entirely different list.

The address of the node to jump to is placed in the **io_Data** field of the I/O request. The new play length/stopping position is taken from this node. However, the new play position is placed in the **io_Offset** field. The drive will seek to the location specified and continue playing audio segments from that point.

Note that the node specified does not need to be within the same list. Thus, you can use this command to switch between segment lists.

This command takes effect and returns immediately.

CDTV_STOPPLAY

This command is used to stop the laser from tracking after a play operation has been aborted with `AbortIO()`. This is done internally by forcing the laser to frame zero, so from a seek standpoint, it is not always a good idea. If you are going to be playing other parts of the disc soon after the `AbortIO()`, this command is not necessary.

This command takes no parameters, and takes effect immediately.

Tips On Playing CD Audio

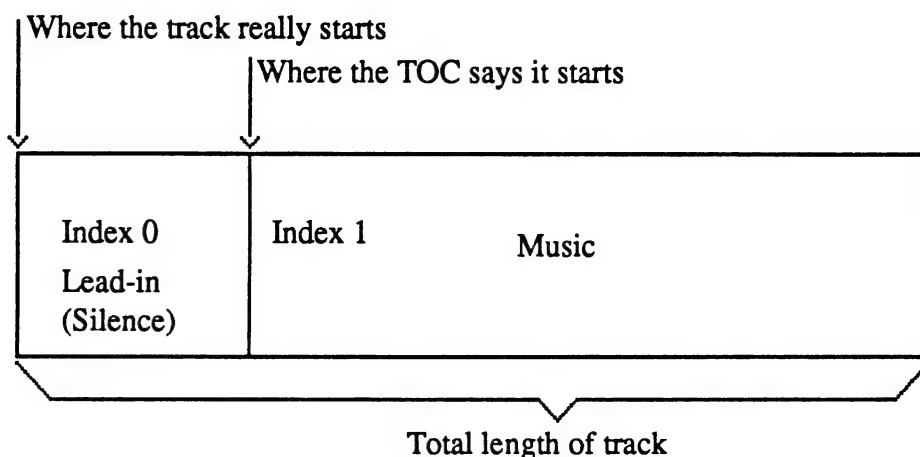
Here are some helpful tidbits on the perils and pitfalls of programming CD audio.

Disc Length

Some people assume that the total length of the disc is the position of the lead-out as reported by the TOC's special disc summary entry. This is not the case. The total usable length of the disc is the location of the lead-out *minus* the starting position of the first track.

Lead-In Segments

Many audio discs have a "lead-in" to each track. This is a zone of total silence before the music actually begins and is technically considered part of the track. However, the TOC will report the beginning of the track where the music actually begins:



When the laser is in the lead-in area, the `SubQ` will report it as being in index 0. Also, the `TrackPosition` field in the `SubQ` report will (in most cases) count down to zero. When actually playing music, it will report it as being in index 1 (or greater, if the disc in question uses indicies), and the `TrackPosition` field will count up. (The `DiskPosition` field always counts up.) Note that not all discs use lead-ins.

If you want to display the `TrackPosition` countdown, you may wish to add one second to the report while the disc is in index zero, since the countdown is unsigned. Adding one second to all `TrackPosition` reports within index zero will prevent you from displaying 00:00 for two seconds.

This is only a real problem if you're planning on creating your own CD audio control panel. This information is presented here so that it doesn't surprise you later on (like it did certain authors).

CDTV_TOC{LSN,MSF}

This command returns the disc's Table Of Contents (TOC), or any portion thereof. The TOC is placed in an array of CDTOC structures. The structure is defined in *<devices/cdtv.h>*, and appears below:

```
struct CDTOC
{
    UBYTE   rsvd;           /* not used      */
    UBYTE   AddrCtrl;       /* SubQ info     */
    UBYTE   Track;          /* Track number  */
    UBYTE   LastTrack;      /* Only for entry zero. */
    CDPOS   Position;       /* Position on disc. */
};
```

The fields in the CDTOC structure can take on two different meanings, depending on how the command is invoked. For ordinary tracks, the fields in the structure have the following meanings:

AddrCtrl

Identical to the AddrCtrl field in the CDSubQ structure. The high four bits may be safely ignored.

Track

The track number this structure describes. This number ranges from 1–99.

LastTrack

Not applicable.

Position

Location of the beginning of the track, in either LSN or MSF format.

When requesting a TOC, you can also request a special extra entry that summarizes the disc. It describes the total number of tracks on the disc, and the total length of the disc. This special entry will be written to the first element in the CDTOC array. For this first entry only, the fields in the CDTOC structure have the following meanings:

AddrCtrl

Not applicable.

Track

First track on this disc. This number is usually 1.

LastTrack

Last track on this disc.

Position

Location of the "lead-out," or end of the disc, in either LSN or MSF format.

The address of the base of the CDTOC array is placed in the *io_Data* field of the I/O request. The number of TOC entries you wish to fetch is placed in *io_Length*; the CDTOC array must have at least this number of elements. The number of the first track whose TOC entry you wish to fetch is placed in *io_Offset*. For example, if you wanted to fetch TOC entries starting with track four, you would place the value 4 in *io_Offset*. If you wish to fetch the special disc summary, you should place the value zero in *io_Offset*.

After completion, if there was no error, the *io_Actual* field will contain the highest track number written to the array (*not* the total number of entries fetched).

Using the CDTV_TOC(LSN,MSF) command might take the following form:

```
struct CDTOC      toarray[100];
LONG              error;
short             firsttrack, lasttrack, numtracks;
short             i;

cdio->io_Command = CDTV_TOCLSN;
cdio->io_Offset  = 0; /* Gimme summary */
cdio->io_Length  = 100; /* Max 99 tracks + summary */
cdio->io_Data    = toarray;
if (error = DoIO (cdio))
    err ("Failed to retrieve TOC.");

firsttrack = toarray[0].Track;
lasttrack  = toarray[0].LastTrack;
numtracks  = lasttrack - firsttrack + 1;
printf ("First track: %d   Last track: %d   Lead-out: %d\n",
        firsttrack, lasttrack, toarray[0].Position);

for (i = 1; i <= numtracks; i++)
    printf ("Track: %2d   AddrCtrl: 0x%02x   Location: %d\n",
            toarray[i].Track,
            toarray[i].AddrCtrl,
            toarray[i].Position.LSN);
```

Miscellaneous Commands

A variety of other useful commands exist:

CDTV_INFO

Retrieves information about the drive.

CDTV_FRAMECALL

Call client code after every N CD frames.

CDTV_FRONTANEL

Enable/disable front panel controls.

CDTV_GENLOCK

Set genlock mode.

CDTV_FRAMECALL

CDTV_FRAMECALL provides a method of receiving regularly paced events synchronized to the CD frame rate. This is accomplished by the device driver calling a programmer-supplied function every *n* CD frames.

A pointer to the function to be called is placed in the `io_Data` field of the I/O request. The number of frames to wait between call-backs is placed in `io_Length`. You invoke this command using `SendIO()`. The driver will hold this command until it is explicitly removed via `AbortIO()`.

The driver will then call your code each time the number of frames specified in `io_Length` have passed. The details of the call-back are covered in the call-back environment section of this article.

CDTV_FRONT PANEL

CDTV's front panel and infrared remote controls can manipulate the audio capabilities of the CD-ROM drive directly, without host (Amiga) intervention. This capability is desirable to "power users" who might wish to play a CD while interacting with a CLI or Workbench. However, certain titles may wish to manipulate CD audio without interference from the user.

This command permits you to turn the front panel controls on and off. When the front panel is "enabled", the front panel controls operate the drive directly, and no rawkey codes for the PLAY, STOP, FF, and REW keys are sent to the computer. When it is "disabled," this direct connection is broken, and the computer receives rawkey codes for those keys.

The desired state of the controls is placed in the `io_Length` field of the I/O request; a 0 disables the controls, a 1 enables them.

The rawkey codes for the CD audio control keys are:

STOP	0x72
PLAY/PAUSE	0x73
FF	0x74
REW	0x75

Watch Out For This Bug. There is a bug in the current hardware which causes these keys to be reported improperly.

For the STOP and PLAY/PAUSE keys, upon depression of the key, the rawkey code is sent (downstroke) and is immediately followed by the same key code with `IECODE_UP_PREFIX` set (upstroke). No code is sent upon key release. These keys do not auto-repeat.

For the FF and REW keys, upon depression of the key, nothing happens. If the key is released immediately, the downstroke code is sent and immediately followed by the upstroke code. If the key is held down, then a rapid repeating series of downstroke- upstroke pairs is sent. The `IEQUALIFIER_REPEAT` bit is not set. The downstroke-upstroke pairs are sent approximately six times a second, and ceases when the key is released.

This bug will be corrected in the future, making the behavior of these keys consistent with the rest of the system. For the FF and REW keys, this corrected behavior may be brought out on a different set of rawkey code numbers.

CDTV_GENLOCK

One of the standard peripherals available for CDTV is a genlock card, which plugs into the back of the machine, replacing the existing multi-format video generator card. The genlock takes an existing video signal and overlays CDTV video on top.

The desired genlock mode is placed in the `io_Offset` field of the I/O request. The available modes are:

CDTV_GENLOCK_AMIGA

Display CDTV-generated video only.

CDTV_GENLOCK_EXTERNAL

Display external video source only.

CDTV_GENLOCK_MIXED

Display CDTV video overlayed on external video.

CDTV_GENLOCK_REMOTE

Allow the infrared remote control to cycle through the above three modes (by using the GENLOCK button).

It is recommended that you return the genlock to CDTV_GENLOCK_REMOTE when your title is finished with it, so the user may regain control of the genlock display mode.

As of this writing, there is no way to detect if a genlock card is installed in CDTV. However, you can detect if the system has come up in genlock mode, which suggests strongly that a genlock is present. This is done by inspecting the DisplayFlags field in GfxBase and seeing if the GENLOC bit is set.

Call-Back Environment

Some *cdtv.device* commands have facilities to call client code. All call-backs occur in the same environment.

Client routines are invoked as a subroutine by the driver, and must return via an RTS instruction. Your code will be running in interrupt mode on the supervisor stack—*under no circumstances should you take advantage of this*. The driver cannot proceed until your code returns, so you must keep it very short; set a variable, send a signal, or cause a software interrupt. Hanging around too long will kill the driver.

CPU registers D0,D1,A0,A1 are scratch. All other registers must be preserved. Each command initializes certain registers in different ways. The initial register contents for each command are as follows:

<u>Command</u>	<u>Registers</u>
CDTV_FRAMECALL	A2: Pointer to I/O request used to invoke the command.
CDTV_PLAYSEGS{LSN,MSF}	A2: Pointer to CDAudioSeg structure just started or completed.
CDTV_READXL	A2: Pointer to CDXL node just completed.

The contents of all other registers are undefined.

Currently, no return codes are recognized by the driver. However, you should set D0 to zero upon exit from your call-back for future compatibility.

See the relevant *Autodoc* page for more details on the call-back environment for a given command.

Example Code Fragments

The following code fragments are further examples of how to call the various *cdtv.device* commands.

```

/*
Directory
-----

Assumptions

CDTV_RESET
CDTV_READ
CDTV_MOTOR
CDTV_SEEK
CDTV_CHANGENUM
CDTV_CHANGESTATE
CDTV_ADDCHANGEINT
CDTV_REMCHANGEINT
CDTV_QUICKSTATUS
CDTV_INFO
CDTV_ISROM
CDTV_FRONT_PANEL
CDTV_FRAMECALL
CDTV_FRAMECOUNT
CDTV_READXL
CDTV_PLAYTRACK
CDTV_PLAYLSN
CDTV_PLAYMSF
CDTV_PLAYSEGSLSN
CDTV_PLAYSEGMSF
CDTV_TOCLSN
CDTV_TOCMSF
CDTV_SUBQLSN
CDTV_SUBQMSF
CDTV_PAUSE
CDTV_STOPPLAY
CDTV_POKESEGSLSN
CDTV_POKESEGMSF
CDTV_MUTE
CDTV_FADE
CDTV_POKEPLAYLSN
CDTV_POKEPLAYMSF
CDTV_GENLOCK
*/

/*****
Assumptions
*****/

{
    struct MsgPort *IOPort;
    struct IOStdReq *IOReq;
    BYTE Err;

    // Init: Create request
    IOPort = CreatePort( 0, 0 );
    if ( ! IOPort )
        Error( NO_PORT );

    IOReq = CreateStdIO( IOPort );
    if ( ! IOReq )
        Error( NO_REQUEST );

    if ( OpenDevice( CDTV_NAME, 0, IOReq, 0 ) )
    {
        IOReq->io_Device = NULL;
        Error( OPEN_DEV );
    }

    // Example code here...

    // Quit: Remove request (Error() must call this)
    if ( IOReq )

```

```

        {
            if ( IOReq->io_Device )
                CloseDevice( (struct IORequest *) IOReq );

            DeleteStdIO( IOReq );
        }

    if ( IOPort )
        DeletePort( IOPort );
}

/*****
CDTV_RESET
*****/

{
    IOReq->io_Command = CDTV_RESET;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_READ
*****/

{
    static UBYTE Buffer[ 2048 + READ_PAD_BYTES ];
    ULONG Sector;
    ULONG Position;
    ULONG Length;

    // - Remember to add READ_PAD_BYTES to read's (and allow
    //   for the extra room in the Buffer).
    Sector = 112;
    Position = ( Sector * 2048 );
    Length = 2048;

    IOReq->io_Command = CDTV_READ;
    IOReq->io_Offset = Position;
    IOReq->io_Length = ( Length + READ_PAD_BYTES );
    IOReq->io_Data = Buffer;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_MOTOR
*****/

{
    ULONG PreviousState;

    IOReq->io_Command = CDTV_MOTOR;
    IOReq->io_Offset = 0;
    IOReq->io_Length = FALSE;    // Turn motor off
    IOReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );

    PreviousState = IOReq->io_Actual;
}

/*****
CDTV_SEEK
*****/

{
    ULONG Position = ( 16 * 2048 );

```

```

        IOReq->io_Command = CDTV_SEEK;
        IOReq->io_Offset = Position;
        IOReq->io_Length = 0;
        IOReq->io_Data = NULL;
        if ( Err = DoIO( (struct IORequest *) IOReq ) )
            printf( "IO Error %ld\n", Err );
    }

/*****
CDTV_CHANGENUM
*****/

{
    IOReq->io_Command = CDTV_CHANGENUM;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );

    printf( "%ld disk changes\n", IOReq->io_Actual );
}

/*****
CDTV_CHANGESTATE
*****/

{
    IOReq->io_Command = CDTV_CHANGESTATE;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );

    if ( IOReq->io_Actual )
        printf( "CD is not in drive\n" );
    else
        printf( "CD is in drive\n" );
}

/*****
CDTV_ADDCHANGEINT
*****/

struct Interrupt ChangeInt;

void __interrupt __saves InterruptCode(
    void
)
{
    // Do something
}

{
    ChangeInt.is_Node.ln_Type = NT_INTERRUPT;
    ChangeInt.is_Code = InterruptCode;

    IOReq->io_Command = CDTV_ADDCHANGEINT;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = (APTR) &ChangeInt;
    SendIO( (struct IORequest *) IOReq );

    // Refer to CDTV_REMCHANGEINT for removal
}

/*****
CDTV_REMCHANGEINT
*****/

struct Interrupt ChangeInt;

```

```

void __interrupt __saveds InterruptCode(
    void
)
{
    // Do something
}

{
    // Refer to CDTV_ADDCHANGEINT for adding
    // ChangeInt

    // (do not need WaitIO()).
    IOREq->io_Command = CDTV_REMCHANGEINT;
    IOREq->io_Offset = 0;
    IOREq->io_Length = 0;
    IOREq->io_Data = (APTR) &ChangeInt;
    DoIO( (struct IORequest *) IOREq );
}

/*****
CDTV_QUICKSTATUS
*****/

{
    IOREq->io_Command = CDTV_QUICKSTATUS;
    IOREq->io_Offset = 0;
    IOREq->io_Length = 0;
    IOREq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOREq ) )
        printf( "IO Error %ld\n", Err );

    // Valid flags:
    if ( IOREq->io_Actual & QSF_AUDIO )
        printf( "Audio playing\n" );

    if ( IOREq->io_Actual & QSF_SPIN )
        printf( "CD is spinning\n" );

    if ( IOREq->io_Actual & QSF_DISK )
        printf( "CD in drive\n" );
}

/*****
CDTV_INFO
*****/

{
    IOREq->io_Command = CDTV_INFO;
    IOREq->io_Offset = CDTV_INFO_FRAME_RATE;
    IOREq->io_Length = 0;
    IOREq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOREq ) )
        printf( "IO Error %ld\n", Err );

    printf( "Frame rate: %ld frames/second\n", IOREq->Actual );
}

/*****
CDTV_ISROM
*****/

{
    IOREq->io_Command = CDTV_ISROM;
    IOREq->io_Offset = 0;
    IOREq->io_Length = 0;
    IOREq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOREq ) )
        printf( "IO Error %ld\n", Err );

    if ( IOREq->io_Actual )
        printf( "CD is a CD-ROM\n" );
    else
        printf( "CD is an audio CD\n" );
}

```

```

/*****
CDTV_FRONTPANEL
*****/

{
    IReq->io_Command = CDTV_FRONTPANEL;
    IReq->io_Offset = 0;
    IReq->io_Length = FALSE;    // to disable front panel
    IReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_FRAMECALL
*****/

void __interrupt __saved __asm InterruptCode(
    register __a2 struct IOStdReq *Request
)
{
    // Do something
    return( 0 );
}

{
    ULONG Frames = ( 2 * 75 );    // every 2 seconds

    IReq->io_Command = CDTV_FRAMECALL;
    IReq->io_Offset = 0;
    IReq->io_Length = Frames;
    IReq->io_Data = (APTR) InterruptCode;
    SendIO( (struct IORequest *) IReq );

    /*
     *
     * Body of code goes here
     *
    */

    // Cleanup:
    // Proper way to terminate
    AbortIO( (struct IORequest *) IReq );
    WaitIO( (struct IORequest *) IReq );
}

/*****
CDTV_FRAMECOUNT
*****/

{
    IReq->io_Command = CDTV_FRAMECOUNT;
    IReq->io_Offset = 0;
    IReq->io_Length = 0;
    IReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IReq ) )
        printf( "IO Error %ld\n", Err );

    printf( "%ld frames since start\n", IReq->io_Actual );
}

/*****
CDTV_READXL
*****/

void __interrupt __saved __asm InterruptCode(
    register __a2 struct CDXL *NodeCompleted
)
{
    // Do something
    return( 0 );
}

```



```

{
static UBYTE Buffer[ 4 ][ 2048 ];
struct MinList CDXLList;
struct CDXL CDXLNode[ 4 ];
ULONG Sector = 16;
ULONG NumSectors = 4;
ULONG i;

// Prepare list
NewList( (struct List *) &CDXLList );

for ( i = 0; i < 4; i++ )
{
AddTail( (struct List *) &CDXLList, (struct Node *) &CDXLNode[ i ] );
CDXLNode[ i ].Buffer = Buffer[ i ];
CDXLNode[ i ].Length = 2048;
CDXLNode[ i ].DoneFunc = InterruptCode;
}

// Start CDXL
IOReq->io_Command = CDTV_READXL;
IOReq->io_Offset = Sector;
IOReq->io_Length = NumSectors;
IOReq->io_Data = (APTR) CDXLList.mlh_Head;
SendIO( (struct IORequest *) IOReq );

/*
 *
 * Body of code goes here
 *
 */

// Cleanup:
// Proper way to terminate
AbortIO( (struct IORequest *) IOReq );
WaitIO( (struct IORequest *) IOReq );

// Then a SEEK is required to force the drive to stop (sometimes necessary)
IOReq->io_Command = CDTV_SEEK;
IOReq->io_Offset = 0; // or next position
IOReq->io_Length = 0;
IOReq->io_Data = NULL;

if ( Err = DoIO( (struct IORequest *) IOReq ) )
printf( "IO Error %ld\n", Err );
}

/*****
CDTV_PLAYTRACK
*****/

{
ULONG StartTrack = 4;
ULONG StopTrack = 6;

IOReq->io_Command = CDTV_PLAYTRACK;
IOReq->io_Offset = StartTrack;
IOReq->io_Length = StopTrack;
IOReq->io_Data = NULL;
if ( Err = DoIO( (struct IORequest *) IOReq ) )
printf( "IO Error %ld\n", Err );
}

/*****
CDTV_PLAYLSN
*****/

{
CDPOS StartSector;
CDPOS SectorsLength;

StartSector.LSN = 16;
SectorsLength.LSN = 300;
}

```

```

IOReq->io_Command = CDTV_PLAYLSN;
IOReq->io_Offset = StartSector;
IOReq->io_Length = SectorsLength;
IOReq->io_Data = NULL;
if ( Err = DoIO( (struct IORequest *) IOReq ) )
    printf( "IO Error %ld\n", Err );
}

/*****
CDTV_PLAYMSF
*****/

{
    CDPOS StartMSF;
    CDPOS StopMSF;

    StartMSF.Raw = TOMSF( 5, 0, 0 );
    StopMSF.Raw = TOMSF( 5, 10, 25 );

    IOReq->io_Command = CDTV_PLAYMSF;
    IOReq->io_Offset = (ULONG) StartMSF;
    IOReq->io_Length = (ULONG) StopMSF;
    IOReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_PLAYSEGSLSN
*****/

void __interrupt __saved __asm InterruptCode(
    register __a2 struct CDAudioSeg *CompletedSeg
)
{
    // Do something
    return( 0 );
}

{
    struct MinList SegList;
    struct CDAudioSeg SegNode;

    // Prepare list
    NewList( (struct List *) &SegList );

    AddTail( (struct List *) &SegList, (struct Node *) &SegNode );
    SegNode.Start.LSN = 16;
    SegNode.Stop.LSN = 300;
    SegNode.StartFunc = InterruptCode;

    IOReq->io_Command = CDTV_PLAYSEGSLSN;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = (APTR) SegList.mlh_Head;
    SendIO( (struct IORequest *) IOReq );

    /*
     * Body of code goes here
     */

    // Cleanup:
    WaitIO( (struct IORequest *) IOReq );
}

/*****
CDTV_PLAYSEGSMSF
*****/

void __interrupt __saved __asm InterruptCode(
    register __a2 struct CDAudioSeg *CompletedSeg
)

```

```

    {
        // Do something
        return( 0 );
    }

    {
        struct MinList    SegList;
        struct CDAudioSeg SegNode;

        // Prepare list
        NewList( (struct List *) &SegList );

        AddTail( (struct List *) &SegList, (struct Node *) &SegNode );
        SegNode.Start.Raw = TOMSF( 5, 0, 0 );
        SegNode.Stop.Raw  = TOMSF( 5, 10, 25 );
        SegNode.StartFunc = InterruptCode;

        IOReq->io_Command = CDTV_PLAYSEGSMF;
        IOReq->io_Offset  = 0;
        IOReq->io_Length  = 0;
        IOReq->io_Data    = (APTR) SegList.mlh_Head;
        SendIO( (struct IORequest *) IOReq );

        /*
         *
         * Body of code goes here
         *
         */

        // Cleanup:
        WaitIO( (struct IORequest *) IOReq );
    }

/*****
CDTV_TOCLSN
*****/

{
    static struct CDTOC TOCArray[ 100 ];
    LONG i;

    IOReq->io_Command = CDTV_TOCLSN;
    IOReq->io_Offset  = 0;                // 0=Summary, or #
    IOReq->io_Length  = 100;
    IOReq->io_Data    = (APTR) TOCArray;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );

    // Disk summary
    printf( "Tracks %d to %d\n",
        TOCArray[ 0 ].Track,
        TOCArray[ 0 ].LastTrack );

    // TOC
    for ( i = 1; i <= CDTVIOReq->io_Actual; i++ )
        printf( "Track %d starts at sector %d\n",
            i,
            TOCArray[ i ].Position.LSN );
}

/*****
CDTV_TOCMSF
*****/

{
    static struct CDTOC TOCArray[ 100 ];
    LONG i;

    IOReq->io_Command = CDTV_TOCMSF;
    IOReq->io_Offset  = 0;                // 0=Summary, or #
    IOReq->io_Length  = 100;
    IOReq->io_Data    = (APTR) TOCArray;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

```

```

// Disk summary
printf( "Tracks %d to %d\n",
        TOCArray[ 0 ].Track,
        TOCArray[ 0 ].LastTrack );

// TOC
for ( i = 1; i <= CDTVIOReq->io_Actual; i++ )
    printf( "Track %d starts at Minute:Second:Frame %d:%d:%d\n",
            i,
            TOCArray[ i ].Position.MSF.Minute,
            TOCArray[ i ].Position.MSF.Second,
            TOCArray[ i ].Position.MSF.Frame );
}

/*****
CDTV_SUBQLSN
*****/

{
    struct CDSUBQ ReqSubQ;
    UBYTE AddrInfo;

    IOReq->io_Command = CDTV_SUBQLSN;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = (APTR) ReqSubQ;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );

    // First check that it is valid
    AddrInfo = ( ReqSubQ.AddrCtrl & ADRCTL_MASK );
    if ( AddrInfo == ADRCTL_NOMODE )
        return; // INVALID SUBQ

    switch ( ReqSubQ.Status )
    {
        case ( SQSTAT_NOTVALID ):
        case ( SQSTAT_NOSTAT ):
            return; // INVALID SUBQ

        case ( SQSTAT_PLAYING ):
            printf( "CD Playing\n" );
            break;

        case ( SQSTAT_PAUSED ):
            printf( "CD Paused\n" );
            break;

        case ( SQSTAT_DONE ):
            printf( "Last Play finished normally\n" );
            break;

        case ( SQSTAT_ERROR ):
            printf( "Last Play finished with error\n" );
            break;
    }

    // Position
    if ( AddrInfo == ADRCTL_POSITION )
    {
        printf( "Current track %d, index %d\n",
                ReqSubQ.Track,
                ReqSubQ.Index );

        printf( "Sector %d into CD, sector %d into track\n",
                ReqSubQ.DiskPosition.LSN,
                ReqSubQ.TrackPosition.LSN );
    }
}

/*****
CDTV_SUBQMSF
*****/

{

```

```

struct CDSUBQ ReqSubQ;
UBYTE AddrInfo;

IOReq->io_Command = CDTV_SUBQMSF;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = (APTR) ReqSubQ;
if ( Err = DoIO( (struct IORequest *) IOReq ) )
    printf( "IO Error %ld\n", Err );

// First check that it is valid
AddrInfo = ( ReqSubQ.AddrCtrl & ADRCTL_MASK );
if ( AddrInfo == ADRCTL_NOMODE )
    return; // INVALID SUBQ

switch ( ReqSubQ.Status )
{
    case ( SQSTAT_NOTVALID ):
    case ( SQSTAT_NOSTAT ):
        return; // INVALID SUBQ

    case ( SQSTAT_PLAYING ):
        printf( "CD Playing\n" );
        break;

    case ( SQSTAT_PAUSED ):
        printf( "CD Paused\n" );
        break;

    case ( SQSTAT_DONE ):
        printf( "Last Play finished normally\n" );
        break;

    case ( SQSTAT_ERROR ):
        printf( "Last Play finished with error\n" );
        break;
}

// Position
if ( AddrInfo == ADRCTL_POSITION )
{
    printf( "Current track %d, index %d\n",
        ReqSubQ.Track,
        ReqSubQ.Index );

    printf( "Minute:Second:Frame %d:%d:%d into CD, %d:%d:%d into track\n",
        ReqSubQ.DiskPosition.MSF.Minute,
        ReqSubQ.DiskPosition.MSF.Second,
        ReqSubQ.DiskPosition.MSF.Frame,
        ReqSubQ.TrackPosition.MSF.Minute,
        ReqSubQ.TrackPosition.MSF.Second,
        ReqSubQ.TrackPosition.MSF.Frame );
}
}

/*****
CDTV_PAUSE
*****/

{
    IOReq->io_Command = CDTV_PAUSE;
    IOReq->io_Offset = 0;
    IOReq->io_Length = TRUE;
    IOReq->io_Data = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_STOPPLAY
*****/

{
    // Remember to AbortIO the current play command first
    IOReq->io_Command = CDTV_STOPPLAY;

```

```

IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = NULL;
if ( Err = DoIO( (struct IORequest *) IOReq ) )
    printf( "IO Error %ld\n", Err );
}

/*****
CDTV_POKESEGLSN
*****/

{
    struct IOStdReq  PokeIOReq;
    struct MinList   SegList;
    struct CDAudioSeg SegNode;
    CDPOS NewSector;

    // Use a copy of our IOReq
    PokeIOReq = *IOReq;

    // Prepare list
    NewList( (struct List *) &SegList );

    AddTail( (struct List *) &SegList, (struct Node *) &SegNode );
    SegNode.Start.LSN = 16;
    SegNode.Stop.LSN = 300;

    IOReq->io_Command = CDTV_PLAYSEGLSN;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = (APTR) SegList.mlh_Head;
    SendIO( (struct IORequest *) IOReq );

    // Actual "poke"
    NewSector.LSN = 480;

    PokeIOReq->io_Command = CDTV_POKESEGLSN;
    PokeIOReq->io_Offset = (ULONG) NewSector;
    PokeIOReq->io_Length = 0;
    PokeIOReq->io_Data = (APTR) SegList.mlh_Head;
    if ( Err = DoIO( (struct IORequest *) PokeIOReq ) )
        printf( "IO Error %ld\n", Err );

    // Cleanup:
    WaitIO( (struct IORequest *) IOReq );
}

/*****
CDTV_POKESEGMSE
*****/

{
    struct IOStdReq  PokeIOReq;
    struct MinList   SegList;
    struct CDAudioSeg SegNode;
    CDPOS NewMSF;

    // Use a copy of our IOReq
    PokeIOReq = *IOReq;

    // Prepare list
    NewList( (struct List *) &SegList );

    AddTail( (struct List *) &SegList, (struct Node *) &SegNode );
    SegNode.Start.Raw = TOMSF( 5, 0, 0 );
    SegNode.Stop.Raw = TOMSF( 5, 10, 25 );

    IOReq->io_Command = CDTV_PLAYSEGLSN;
    IOReq->io_Offset = 0;
    IOReq->io_Length = 0;
    IOReq->io_Data = (APTR) SegList.mlh_Head;
    SendIO( (struct IORequest *) IOReq );

    // Actual "poke"

```

```

NewMSF.Raw          = TOMSF( 6, 10, 20 );

PokeIOReq->io_Command = CDTV_POKESEGMFSF;
PokeIOReq->io_Offset  = (ULONG) NewSector;
PokeIOReq->io_Length  = 0;
PokeIOReq->io_Data    = (APTR) SegList.mlh Head;
if ( Err = DoIO( (struct IORequest *) PokeIOReq ) )
    printf( "IO Error %ld\n", Err );

// Cleanup:
WaitIO( (struct IORequest *) IOReq );
}

/*****
CDTV_MUTE
*****/

{
    IOReq->io_Command = CDTV_MUTE;
    IOReq->io_Offset  = 0;           // Volume 0-0x7FFF
    IOReq->io_Length  = 1;           // 0=check, 1=next play stop, 2=now
    IOReq->io_Data    = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_FADE
*****/

{
    ULONG Seconds      = 2;

    IOReq->io_Command = CDTV_FADE;
    IOReq->io_Offset  = 0x7FFF;      // Volume 0-0x7FFF
    IOReq->io_Length  = ( Seconds * 75 );
    IOReq->io_Data    = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

/*****
CDTV_POKEPLAYLSN
*****/

{
    struct IOStdReq PokeIOReq;
    CDPOS StartSector;
    CDPOS SecondSector;
    CDPOS SectorsLength;

    // Use a copy of our IOReq
    PokeIOReq = *IOReq;

    StartSector.LSN = 16;
    SectorsLength.LSN = 300;

    IOReq->io_Command = CDTV_PLAYLSN;
    IOReq->io_Offset  = StartSector;
    IOReq->io_Length  = SectorsLength;
    IOReq->io_Data    = NULL;
    SendIO( (struct IORequest *) IOReq );

    // Actual "poke"
    SecondSector.LSN = 480;

    PokeIOReq->io_Command = CDTV_POKEPLAYLSN;
    PokeIOReq->io_Offset  = SecondSector;
    PokeIOReq->io_Length  = SectorsLength;
    PokeIOReq->io_Data    = NULL;
    if ( Err = DoIO( (struct IORequest *) PokeIOReq ) )
        printf( "IO Error %ld\n", Err );

    // Cleanup:

```

```

    WaitIO( (struct IORequest *) IOReq );
}

/*****
CDTV_POKEPLAYMSF
*****/

{
    struct IOStdReq PokeIOReq;
    CDPOS StartMSF;
    CDPOS SecondMSF;
    CDPOS StopMSF;

    // Use a copy of our IOReq
    PokeIOReq = *IOReq;

    StartMSF.Raw      = TOMSF( 5, 0, 0 );
    SecondMSF.Raw     = TOMSF( 5, 40, 0 );

    IOReq->io_Command = CDTV_PLAYMSF;
    IOReq->io_Offset  = (ULONG) StartMSF;
    IOReq->io_Length  = (ULONG) StopMSF;
    IOReq->io_Data    = NULL;
    SendIO( (struct IORequest *) IOReq );

    // Actual "poke"
    StopMSF.Raw      = TOMSF( 5, 10, 25 );

    PokeIOReq->io_Command = CDTV_POKEPLAYMSF;
    PokeIOReq->io_Offset  = (ULONG) SecondMSF;
    PokeIOReq->io_Length  = (ULONG) StopMSF;
    PokeIOReq->io_Data    = NULL;
    if ( Err = DoIO( (struct IORequest *) PokeIOReq ) )
        printf( "IO Error %ld\n", Err );

    // Cleanup:
    WaitIO( (struct IORequest *) IOReq );
}

/*****
CDTV_GENLOCK
*****/

{
    IOReq->io_Command = CDTV_GENLOCK;
    IOReq->io_Offset  = CDTV_GENLOCK_MIXED;
    IOReq->io_Length  = 0;
    IOReq->io_Data    = NULL;
    if ( Err = DoIO( (struct IORequest *) IOReq ) )
        printf( "IO Error %ld\n", Err );
}

```

1

2

3

Bookmark and Cardmark Device Drivers

Overview

Bookmarks provide a means of storing CDTV application data across machine resets and power button shut-downs. For the vast majority of CDTV machines in the world (those not equipped with a floppy disk or hard disk drive), bookmarks serve as the only technique available to program designers for semi-permanent data storage.

Bookmarks were designed primarily to hold tiny hints of data that could be utilized by an application to return the user to a previously marked position within the program (hence the name "Bookmark"). For example, an encyclopedia may provide bookmarks to help users locate commonly referenced topics; a cookbook may indicate the previous recipe prepared; a math tutorial may recall the last lesson successfully completed; and a game may store its top five scores.

There is, of course, a limit to how many bookmarks can be stored. They should not be thought of nor used as a substitute for the larger, permanent storage medium of floppy disk.

Bookmark Memory

Every CDTV unit comes factory equipped with a small module of special memory for the storage of semi-permanent data. This memory is not located in the normal main memory spaces and is not configured into the operating system memory allocation lists. Instead, it has its own private location which is accessed only through the proper software protocols.

Bookmark memory is a "line-backed" RAM rather than battery backed RAM. It will persist and hold its data only while AC power is connected to the unit. The CDTV power switch will not affect the memory, but unplugging the unit or a power failure will destroy its contents. (The front panel clock display serves as a power indicator for the bookmark RAM.) In the current hardware, there is no provision for a capacitive power reserve to carry across momentary power failures or the relocation of the unit to some other room.

Device Driver

The Bookmark Device Driver is responsible for managing all accesses to the bookmark memory. It resides in the Operating System ROM of every CDTV and operates in a standard fashion as an Amiga kernel level (Exec) device driver. The device driver was created with three purposes in mind:

- Managing the limited amount of memory available.
- Sharing of the memory between multiple applications executing concurrently or during different sessions.

- Removing old bookmarks in favor of new ones.
- Hiding the location and size of the memory so that it may be altered or expanded in future hardware designs.

The device driver organizes memory to appear as a "limitless" pool of independent bookmarks. It provides a sort of "file system" that allows multiple applications to store small data segments in a moderately simple way. The driver is not a standard file system however. During its design stage, the decision was made not to use an Amiga file system (e.g., RAM disk) because it was more important to optimize the storage capacity available to applications rather than optimize the access mechanism. As a result, the storage overhead of a bookmark is low thus more bookmarks can be saved.

Cardmarks

In addition to the factory equipped bookmark RAM, CDTV also provides an alternate storage mechanism: credit card style memory cards. These cards allow consumers to expand the memory capabilities of their units in a number of ways. The cards can be used for:

- Expanded bookmark storage (usually battery backed).
- ROM based Operating System enhancements (libraries and devices).
- ROM based program applications.
- Expansion RAM for increased main memory.
- Recoverable RAM disks (optionally battery backed).
- Diagnostic testing.
- Special hardware enhancements.

The *bookmark.device* driver has the capability to access these cards when expanded bookmark storage is desired. They can be formatted for use with bookmarks (often called cardmarks) and managed in a fashion identical to that used with bookmarks.

Applications have major advantages when it comes to storing data in cardmarks. The card's memory is normally much larger than the line-backed bookmark memory, which means a greater number of bookmarks can be saved, and each bookmark can be larger in size. Also, memory cards are usually backed up with battery power, thus protecting them from loss of power.

There is, however, one serious drawback to a memory cards: they can be removed from the machine. Just because your program wrote a cardmark does not mean you will always be able to access it. Also, a user may select an older memory card containing out of date cardmarks. If not dealt with properly in your application, this could lead to consumer confusion. In some cases you may want to create a date stamp in your data.

One possible approach for using cardmarks is to first check to see if a card is available. If so, use it, otherwise use a normal bookmark. It is a good idea to tell the user that a "bookmark is being placed" on the card, so that he can be aware that the card is being used and note what card it is.

Principles

This section describes the general principles of bookmarks and the *bookmark.device* driver. These principles hold for both bookmarks and cardmarks created with the CDTV *bookmark.device*.

Initialization

When CDTV is started for the first time after a full power loss, its line-backed bookmark memory is configured and initialized. The memory is scanned to determine its size, it is cleared to zero, and a header is placed within it, marking that it is free. The *bookmark.device* driver is initialized and becomes available as a standard Exec device.

If a memory card is present, it is examined to determine whether it is already being used for bookmarks, free to be used for bookmarks, or not available for use. If the card is being used for bookmarks, its device driver is initialized. If the card is free, it is scanned to determine its size, cleared to zero, set up for bookmarks, and its driver is initialized. If a card is not available, it is left unused, and the *cardmark.device* driver is not initialized.

Identifiers

In order for multiple applications to access their own specific bookmarks, each bookmark must contain an identifier to distinguish it from the others. Each bookmark is labeled with a unique code called a Bookmark Identifier or BID. A BID must be used to create, read, write, and delete a specific bookmark.

A BID is made up of two parts: a manufacturer code and a product number. The manufacturer code identifies the company or organization to which the bookmark belongs. These codes are issued by Commodore to anyone wishing to develop for CDTV. There are 65,536 (64K) codes available. We can only hope that we will someday run out. To obtain such a code you must call CATS. Please do not make up your own codes—this can have a bad effect on consumers. There are a lot of other more interesting challenges in the world.

The first few manufacturer codes are reserved for Commodore. For example, should you want to access the CDTV preferences, you will use a Commodore code of 0001. Other Commodore codes are used for testing, BID expansion, memory expansion, examples, etc.

The product number identifies the application related to the bookmark. It is normally just a product number created by you. There are 64K possibilities, so you are limited to making only 65,535 applications. At that point you can retire or start a new company and get a new manufacturer code. It is sometimes useful to divide the product number to allow multiple bookmarks for a single product. For example, the upper twelve bits may be what you use to identify your product and the lower four bits may indicate the bookmark number.

The actual form of a BID is specified in the *bookmark.h* file as:

```
structure BookmarkID
{
    UWORD Manufacturer;
    UWORD Product;
};
```

A macro called *MAKEBID* is also defined for creating BID constants.

Content

The content of your bookmark may take any form needed by your application. You may want to use it for storing sector/page/screen/button/line numbers to indicate a return point, a hash value that determines where to resume or what you were doing, character strings, path/directory/file names, high-scores, etc.

The guiding rule is *make whatever you store as small as possible*. With limited space in the bookmark memory, the smaller the average bookmark, the more will fit. If you can get by with just a few bytes, do so. For example, don't waste space by storing 32 bit numbers that only range from one to ten. Chop them down to at least a byte in size.

Another way to reduce bookmark size is to use various index, offset, and table approaches. For example, if you need to save four text strings out of a set of 4000, you could save space by placing the strings in a file, and saving only the file byte offsets in the bookmark. Better yet, store an index into a table of offsets. If you need to store a set of file names, make your file names numeric and store them as binary in a bookmark. If you are writing a game that can be saved in one of 25000 states and you have 50 MBytes free on your CD, you could save the index in a bookmark, and use the CD to store the actual state information. Finally, if you must save textual words, save tokens. You might even want to use the approach of letting your most common 127 words be stored as bytes, and the remaining set stored as 16 bit words.

Bookmarks over a certain size may not be storable within bookmark memory. The device driver will not accept a bookmark that is larger than 1/16 the total size of the bookmark RAM. To determine the maximum size allowed, use the driver's MAXSIZE command.

Aging

Over time it is likely that the CDTV consumer is going to fill his bookmark memory to the point where no additional bookmarks will fit. When this happens, the device driver will start removing older bookmarks in order to make more space. To the consumer, it just looks like the machine forgot something done "a while back". This is a natural way for people to think about it.

Two factors determine what bookmark will be removed: priority and age. The priority of a bookmark is assigned by the programmer when it is created and can be modified later. It can range from -128, a very low priority, to +127, the highest priority. The default or standard value is zero. Bookmarks with a higher priority will remain over a longer period relative to other bookmarks.

You may have a difficult time deciding the priority for your bookmarks. In general, consider the consumer's interest and use the lowest priority possible. By application categories, the following values are suggested:

+40	Education
+20	Reference
0	Arts and Leisure
-20	Music
-40	Entertainment

What's the logic here? Just because Timmy plays twenty-seven games in an evening doesn't mean that he should blow away Dad's Car Repair Encyclopedia, Mom's Italian Language Lesson, or Grandma's Blueberry Pie Recipe (especially that). Think about who bought the machine.

Of course, every developer is naturally going to think that her application is more important than the others. That approach is not going to work. If all priorities are set to 60, the net effect is the same as everyone setting it to zero. When in doubt, use zero.

Another approach is to let the user decide. If you present a menu for saving bookmarks, you might allow the user to decide if the item is "very important" or "not so important". Don't give 256 priority choices, just a few to keep it simple.

As a bookmark remains in memory, it ages. The age is an indication of how long it has been hanging around unreferenced. As a bookmark ages, it becomes more likely that it will be removed to make room for newer bookmarks.

Whenever a bookmark is referenced through a read or a write command, its age is reset. This causes a bookmark to be "reborn" giving it an extended life. This seems like a natural approach, as the most recently used applications should have the "youngest" bookmarks.

Program Access

Access to bookmarks is achieved through the standard Exec Device Driver interface. The device driver is accessed by calling the Exec Library function `OpenDevice()`. For Bookmark RAM, the name "bookmark.device" is used; for Cardmark RAM use "cardmark.device". The BID is passed in the Unit number parameter to `OpenDevice()`.

```
BYTE error;
```

```
error = OpenDevice("bookmark.device", MY_BID, IOReq, 0);
```

This must be done before attempting to do any operations with the *bookmark.device* driver. You always set the BID whether the bookmark exists or not. When a bookmark with the given BID does exist, the IOReq will be linked to it through a couple of its internal fields.

When accessing a bookmark in any way, it is a good idea to open the bookmark, make changes, and then close it. Do not leave it open for the full length of your application.

PROGRAMMERS BEWARE

1. Bookmarks are garbage collected and relocated from time to time. Do not attempt to access a bookmark directly in memory, it may move on you.
2. Both the size and location of the Bookmark RAM will change in coming months as CDTV is improved. Your application will remain unaffected so long as it only accesses bookmarks through the specified device driver.

Once a bookmark has been opened, you can read it, over-write it, update it, clear it, etc. The details of these operations will be discussed in the "Programming" section and in the "Reference" section.

Programming

This section describes the details of programming to obtain the best results from the bookmark and cardmark devices. It will explain the primary approach for using the device driver, creating and accessing bookmarks, as well special information on initializing and formatting new memory cards.

Device Driver

The *bookmark.device* is a standard Exec style device driver as documented in Chapter 19 of the V1.3 *Amiga ROM Kernel Reference Manual: Libraries and Devices*. All device commands are performed through I/O requests sent to the device and all of these commands are executed synchronously (there are no asynchronous commands). The device also supports the formatting and initialization of other memory devices such as memory cards.

The *bookmark.device* driver resides in the Operating System ROM of every CDTV. It is referenced by programs through its device node name "bookmark.device" or "cardmark.device" depending on the type of storage desired. As is the accepted practice for Exec device drivers, all characters in the device names must be lower case.

The *bookmark.device* driver operates with a standard I/O request structure `IOStdReq` as found in the header file *exec/io.h*. You can locate this structure within your application's program data segment, or you can allocate it in Chip RAM (There is no Fast RAM in CDTV units, though this may change in the future). It must be properly initialized before being used (See examples in the RKM and below). You can, of course, reuse the same I/O request structure as much as you need.

The sequence to initialize a typical I/O request might look like:

```
struct MsgPort *IOPort;    /* pointer to the message port */
struct IOStdReq *IOReq;    /* pointer to the I/O request */

IOPort = CreatePort(0,0); /* create the message port */

if (IOPort == NULL)       /* call Error() if port was not created */
    Error(NO_PORT);

IOReq = CreateStdIO(IOPort); /* create the I/O request */

if (IOReq == NULL)        /* call Error() if request was not created */
    Error(NO_REQUEST);
```

The `Error()` function and error constants are not part of the system, they are for you to provide. Normally they would free and close any resources, print a message to the user, and exit.

Once initialized you can open the device with the `OpenDevice()` function. This finds the device driver and binds it to the request:

To open the *bookmark.device*:

```
if (OpenDevice("bookmark.device", MY_BID, IOReq, 0)) /* open the device */
    Error(OPEN_DEV);
```

To open the *cardmark.device*:

```
if (OpenDevice("cardmark.device", MY_BID, IOReq, 0))
    Error(OPEN_DEV);
```

As shown, the unit number (second parameter) contains the Bookmark Identifier (BID). The flags (last parameter) should be zero for future compatibility.

If the device cannot be opened, an error is returned. This happens if the device driver for a particular type of memory cannot be found. The *bookmark.device* should always open (unless the memory is defective). The *cardmark.device* will only open if there is a valid memory card (See section on "Initializing Cards" below).

The requests are now ready to be put to use for I/O. The examples in the rest of this chapter will assume that the device has been opened as shown above. When finished with the requests, they should be passed to `CloseDevice()` before being deleted or freed.

Performing Commands

Bookmark device commands are performed in exactly the same fashion as other Exec devices. The `DoIO()` function will perform synchronous commands, that is, it will not return until the command has completed. The `SendIO()` function also works, but provides no extra advantage because of the synchronous nature of the device.

Before performing a command, the I/O request structure must be set up with the various parameters of the command. For instance, a read might be set up in the following manner:

```
IOReq->io_Offset = 0;          /* start at byte 0 */
IOReq->io_Length = -1;         /* read to end of bookmark*/
IOReq->io_Data = Buffer;       /* put them in Buffer */
IOReq->io_Command = CMD_READ;  /* read */
```

The `io_Command` field always indicates the command to execute. The command constants are supplied in the header file *bookmark.h*. The `io_Offset`, `io_Length`, and `io_Data` fields will vary depending on the command. Normally, the `io_Offset` contains the starting point for a command. In the case of the `CMD_READ` above, it holds the starting byte number. The `io_Length` contains either the size of the bookmark, or as in the case above, -1, indicating NULL-termination. The `io_Data` is used in only a few commands when either a data buffer is needed or additional parameters are required. It should be set to NULL for all other commands.

SPECIAL NOTE ABOUT `io_Length`. In the current version of the *bookmark.device*, the `io_Length` field must be set to either the length of the bookmark you are accessing (as set for the `BD_CREATE` command that created the bookmark), or -1. This means you may only do partial reads and writes from an offset to the end of the bookmark, not from the beginning (or an offset) to a point before the end of the bookmark. Future versions of the device will probably change this behavior.

To perform a synchronous command, pass the request to `DoIO()`:

```
DoIO(IOReq);
```

The function will return when the command has completed. If an error occurs in the command, it will be returned in the `io_Error` field of the request. A zero indicates no error. The error value is also returned from `DoIO()` so you may want to take advantage of it:

```
if (DoIO(IOReq))
    Error(IO_FAILED);
```

The actual error value will depend on the command executed. See the header file *bookmark.h* for error values.

For several commands, there is additional information in the `io_Actual` field of the I/O request. This information is dependent on the command executed. Normally it is related to the value placed in the `io_Length` field of the request. For example, a `CMD_READ` would return the total number of bytes read.

Aborting Commands

Commands cannot be aborted because the *bookmark.device* driver only processes them in a synchronous fashion.

Commands

All of the *bookmark.device* driver commands are defined in the *bookmark.h* header file. From these commands the standard set used by CDTV applications are:

Command	Num	Operation
CMD_READ	2	Read data from bookmark.
CMD_WRITE	3	Write data to bookmark.
CMD_UPDATE	4	Reset bookmark age.
CMD_CLEAR	5	Clear bookmark contents.
BD_CREATE	13	Create new bookmark.
BD_DELETE	14	Delete bookmark.
BD_MAXSIZE	15	Return maximum bookmark size.
BD_SIZEOF	17	Return the size of a bookmark.
BD_SETPRI	18	Set the priority of a bookmark.

The remaining commands are designed for special purposes and would not normally be a part of applications.

Some of these commands are helpful for debugging and testing your applications, and should be removed before your final release. A few other commands are meant to be used by memory card vendors as a means of formatting new memory cards. Specialized OEMs for vertical applications may also use these commands.

Command	Num	Operation
CMD_RESET	1	Reset the bookmark memory.
BD_TYPEMEM	9	Return RAM type.
BD_SIZEMEM	10	Return RAM size (destructive).
BD_INITMEM	11	Initialize RAM.
BD_CREATEDEV	12	Create new device driver.
BD_AVAIL	16	Return number of bytes free in memory.
BD_CHECK	19	Checksum bookmark memory.
BD_PURGE	20	Purge bookmark memory.
BD_DUMP	21	Dump bookmark memory to main memory.
BD_LOAD	22	Load bookmark memory from main memory.

Creating a Bookmark

Creating a new bookmark is easy and is absolutely necessary before the bookmark can be used for writing, reading, or other operations.

To create a bookmark simply follow these steps:

1. Contact Commodore to obtain a manufacturer ID number. You must do this if you want to use bookmarks in your applications, and Commodore has agreed to give them out without a fuss. So for this example *only*, if your number is 0000, you might create a bookmark identifier of:

```
#define MY_BID 0x00000001
```

2. Determine what you are going to store. Use a data structure if you so desire:

```
struct MyBookmark
{
    UWORD SectorNum;
    UBYTE PathNum;
    UBYTE UserOptions;
};
```

3. Calculate the maximum size of your bookmark. If it is a structure you should be able to get by with `sizeof()` in most cases. If you have variable length bookmarks, you will need to know the maximum length.

```
#define MAX_BOOKMARK sizeof(struct MyBookmark)
```

4. Create the I/O reply port and the I/O request:

```
struct MsgPort *IOPort;
struct IOStdReq *IOReq;

IOPort = CreatePort(0,0);

if (IOPort == NULL)
    Error(NO_PORT);

IOReq = CreateStdIO(IOPort);

if (IOReq == NULL)
    Error(NO_REQUEST);
```

5. Open the *bookmark.device* driver. Use your own BID even though it should not exist yet :

```
if (OpenDevice("bookmark.device", MY_BID, IOReq, 0))
    Error(OPEN_DEV);
```

6. Determine and set the aging priority of your bookmark.

```
IOReq->io_Node.ln_Pri = -10;
```

7. Set up the parameters to the create command:

```
IOReq->io_Offset = MY_BID;
IOReq->io_Length = MAX_BOOKMARK;
IOReq->io_Data = 0;
IOReq->io_Command = BD_CREATE;
```

8. Now you are ready for action. Create the bookmark:

```
DoIO (IOReq);
```

9. Do not assume that it worked. Check for errors:

```
switch (IOReq->io_Error)
{
    case 0:
        msg = NULL;
        break;

    case BDERR_TOO_BIG:
        msg = "bookmark is larger than allowed";
        break;

    case BDERR_EXISTS:
        msg = "bookmark with this identifier already exists";
        break;

    case BDERR_NO_SPACE:
        msg = "no space is available for bookmark";
        break;
}

if (msg)
{
    printf("ERROR IN CREATE: %s\n", msg);
    Handle error
}
```

That's it. If there were no errors you know have a bookmark. The command actually allocated the specified amount of space within the bookmark RAM, pushing older bookmarks out if necessary.

These Things Take Time. Be prepared for a short delay (milliseconds) from time to time. The bookmark memory space may need to be compacted if one or more of the older bookmarks must be deleted. This operation is performed during the BD_CREATE command. The length of the delay is variable depending on the number, size and positions of bookmarks. Also larger card memories will have longer delays.

It is important to realize that even though a BID is used in the call to `OpenDevice()`, bookmark memory will not be associated with the request until the create has been done.

Don't forget to close the *bookmark.device* with `CloseDevice()` when you are finished with it.

Writing and Reading

Once a bookmark exists, you can use the standard Exec I/O commands for writing its contents and reading it back.

To write a bookmark:

1. Use the same BID and data structure you developed earlier in the create operation.
2. Create the I/O reply port and the I/O request:

```
struct MsgPort *IOPort;
struct IOStdReq *IOReq;

IOPort = CreatePort(0,0);

if (IOPort == NULL)
    Error(NO_PORT);
```

```
IOReq = CreateStdIO(IOPort);
if (IOReq == NULL)
    Error(NO_REQUEST);
```

3. Open the *bookmark.device* driver. Use your BID here:

```
if (OpenDevice("bookmark.device", MY_BID, IOReq, 0))
    Error(OPEN_DEV);
```

4. Put data into your bookmark structure:

```
struct MyBookmark MB;
MB.SectorNum = Sector;
MB.PathNum = Path;
MB.UserOptions = Options;
```

5. Set up the parameters to the write command:

```
IOReq->io_Offset = 0;
IOReq->io_Length = -1;
IOReq->io_Data = &MB;
IOReq->io_Command = CMD_WRITE;
```

6. Write the bookmark:

```
DoIO(IOReq);
```

7. Check for errors:

```
switch (IOReq->io_Error)
{
    case 0:
        msg = NULL;
        break;

    case BDERR_NOMARK:
        msg = "bookmark does not exist";
        break;

    case BDERR_PASTEND:
        msg = "attempt to write more than allowed";
        break;

    case BDERR_BADARG:
        msg = "bad I/O argument was passed";
        break;
}

if (msg)
{
    printf("ERROR IN WRITE: %s\n", msg);
    Handle error
}
else
    printf("%d BYTES WRITTEN\n", IOReq->io_Actual);
```

If the *io_Offset* has a value other than zero, it is used as a byte offset into the bookmark. This allows you to change subfields of the bookmark independently.

After the command, the *io_Actual* field indicates the number of bytes written.

Reading A Bookmark

To read a bookmark, use the same steps 1, 2, and 3, skip step 4, then:

5. Set up the read command:

```
IOReq->io_Offset = 0;
IOReq->io_Length = -1;
IOReq->io_Data = &MB;
IOReq->io_Command = CMD_READ;
```

6. Read the bookmark:

```
DoIO(IOReq);
```

7. Check for errors. Use the same approach as above.

Deleting A Bookmark

If your bookmark is no longer needed nor desired, remove it from memory to make room for other applications.

To delete a bookmark:

1. Use the same BID and data structure you developed earlier in the create operation.
2. Create the I/O reply port, the I/O request, and open the device as shown in the previous examples, then
5. Set up the parameters to the delete command:

```
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
IOReq->io_Command = BD_DELETE;
```

6. Delete the bookmark:

```
DoIO(IOReq);
```

7. Check for errors:

```
switch (IOReq->io_Error)
{
    case 0:
        msg = NULL;
        break;

    case BDERR_NOMARK:
        msg = "bookmark does not exist";
        break;
}

if (msg)
{
    printf("ERROR IN DELETE: %s\n", msg);
    Handle error
}
```

Creating Cards

Card memory vendors or publishers may in some cases need to devise a means for initializing new memory cards and prepare them for either bookmark or non-bookmark use.

Don't Use This Code. It should be stressed that the code in this section is *not* to be used in CDTV applications. If you do, expect your program to break in a few months when the hardware is revised. It has been included because you may be able to use it for limited types of testing

Memory cards are classified into the following types. The first column is the value (hex or ASCII) present in the first word of the memory.

BK	bookmark initialized RAM
RW	expansion RAM
RD	recoverable RAM disk
RO	special system/application ROM
OK	reserved for something else
\$1111	system diagnostics

When the system bootstraps, the *bookmark.device* examines this memory to determine its type. If it is BK, it assumes the memory contains valid bookmarks. If it is any of the other types, it ignores the card, otherwise it initializes the memory for new bookmarks.

If you are selling a card to be used as expansion RAM or a special RAM disk you need to preset your card to RW or RD. There is another way to do this safely from a utility CD or floppy, but it is beyond the scope of this document.

To create a memory card with preset bookmarks takes a few special commands. The correct sequence is:

1. Define the location of the card memory and its maximum size. Do not use this in CDTV applications!

```
#define CARD_MEM 0xE00000
#define MAX_SIZE 0x080000
```

2. Create the I/O reply port and the I/O request:

```
struct MsgPort *IOPort;
struct IOStdReq *IOReq;

IOPort = CreatePort(0,0);

if (IOPort == NULL)
    Error(NO_PORT);

IOReq = CreateStdIO(IOPort);

if (IOReq == NULL)
    Error(NO_REQUEST);
```

3. Open the *bookmark.device* driver using a zero BID:

```
if (OpenDevice("bookmark.device", 0, IOReq, 0))
    Error(OPEN_DEV);
```

4. Check the type of memory:

```
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = CARD_MEM;
IOReq->io_Command = BD_TPEMEM;
DoIO(IOReq);

if (IOReq->io_Actual) /* Is it available or not? */
{
    puts("Card memory already in use");
    Handle error;
}
```

5. Find the size of the memory. This is a destructive test so use it with care.

```
IOReq->io_Offset = 0;
IOReq->io_Length = MAX_SIZE;
IOReq->io_Data = CARD_MEM;
IOReq->io_Command = BD_SIZEMEM;
DoIO(IOReq);

if (IOReq->io_Actual == 0) /* Is it there? */
{
    puts("Memory card is not present");
    Handle error;
}
```

6. Initialize the memory, getting it ready for bookmarks:

```
IOReq->io_Offset = 0;
IOReq->io_Length = IOReq->io_Actual; /* the size */
IOReq->io_Data = CARD_MEM;
IOReq->io_Command = BD_INITMEM;
DoIO(IOReq);
```

7. At this point you can reset your machine or initialize the *cardmark.device* driver. In theory, it would probably be better to reset your machine, but if this creates problems for your initializing program, you can start the *cardmark.device* driver with the code below.

```
/* Allocate device name in memory */
name = AllocMem(strlen("cardmark.device")+1,0);

if (!name)
    HandleError(...); /* don't return */

strcpy(name, "cardmark.device");

IOReq->io_Offset = name;
IOReq->io_Length = 0;
IOReq->io_Data = CARD_MEM;
IOReq->io_Command = BD_CREATEDEV;
DoIO(IOReq);

if (!IOReq->io_Actual)
    HandleError("cannot initialize cardmark device");
```

The astute programmer may recognize that it is possible to create bookmark devices anywhere in memory. All you need to do is perform an `AllocMem()` then pass the memory address on to `BD_INITMEM`, then `BD_CREATEDEV`. This is one way to simulate the operation of bookmarks without actually accessing the bookmark or cardmark memories. Again, this should only be done for testing purposes and not applications.

Testing Hints

Finally, if your application makes extensive use of bookmarks, you may want to use a few special commands to make your testing easier.

The bookmark and cardmark memories as a whole or in part can be transferred to and from main memory. This allows you to create programs that load an entire set of bookmarks before a test begins. See the reference section for details on the BD_DUMP and BD_LOAD commands.

Also, it is a wise idea to verify before application release that your bookmark memory is free from wild pointer hits and corruption. To do this, do the BD_CHECK command at the start of your program, then again every so often. Set the io_Offset field TRUE when you first do the command and after every bookmark command that modifies the memory. Whenever you receive a non-zero result, something has modified bookmark memory! See the reference section for more information.

Bookmark Device Driver Command Reference

CMD_RESET

Reset the bookmark/cardmark memory to its initial, cleared power-on state. (For testing purposes only.)

Inputs

```
io_Command= CMD_RESET;
io_Offset = 0;
io_Length = 0;
io_Data = 0;
```

Outputs

```
ErrorCode = io_Error;
```

Description

This command resets bookmark or cardmark memory to its initial configuration state. The entire memory is cleared and all marks are lost. This operation is not suggested for released programs, but can be used for application testing when it is necessary to clear mark memory to a known state.

Note that for this command to work correctly, the device memory header must be intact. This command is not to be used to initialize new, blank, or unformatted memory devices (see BD_INITMEM).

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
IOReq->io_Command = CMD_RESET; /* That's all folks */
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
DoIO(IOReq);
```

Errors

None

Related Commands

BD_INITMEM

BD_CREATEDEV

CMD_READ

Read data from an existing bookmark.

Inputs

```
io_Command= CMD_READ
io_Offset = ByteOffset
io_Length = NumberOfBytes
io_Data = Buffer
```

Outputs

```
ErrorCode = io_Error
BytesRead = io_Actual
```

Description

Transfer the bookmark into memory. The bookmark must already exist (BD_CREATE) and contain data (CMD_WRITE).

If a -1 is specified for the length, the entire remaining contents of the bookmark are transferred.

The actual number of bytes read is returned in io_Actual.

If io_Offset + io_Length exceeds the size of the bookmark, a BDERR_PASTEND occurs and no data is read.

Each time a bookmark is read, its age is reset, extending the life of the bookmark in memory.

WARNING: Never attempt to access a bookmark directly in bookmark/cardmark memory space. Bookmark space is compacted from time to time and bookmarks may be relocated without your knowledge.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
char Buffer [52];

IOReq->io_Command = CMD_READ;
IOReq->io_Offset = 0;
IOReq->io_Length = 52;
IOReq->io_Data = Buffer;
if (DoIO(IOReq)) ProcessError(IOReq);
    printf("%d bytes read/n", IOReq->io_Actual);

IOReq->io_Command = CMD_READ;
IOReq->io_Offset = 10;
IOReq->io_Length = -1;
IOReq->io_Data = Buffer;
if (DoIO(IOReq)) ProcessError(IOReq);
    printf("%d bytes read/n", IOReq->io_Actual);
```

Errors

NOMARK	no bookmark exists
PASTEND	attempt read more than possible
BADARG	bad I/O argument

Related

CMD_WRITE
BD_CREATE

CMD_WRITE

Write data to an existing bookmark.

Inputs

```
io_Command= CMD_WRITE
io_Offset = ByteOffset
io_Length = NumberOfBytes
io_Data   = Buffer
```

Outputs

```
ErrorCode = io_Error
BytesRead = io_Actual
```

Description

Transfer from memory into the bookmark. The bookmark must already exist or have been created with BD_CREATE.

The maximum size of the transfer is limited to the maximum size of the bookmark as specified in the original create operation. If a -1 is specified for the length, the full bookmark size is used for the transfer.

The actual number of bytes written is returned in io_Actual.

If io_Offset + io_Length exceeds the size of the bookmark, a BDERR_PASTEND occurs and no data is written.

Each time a bookmark is written, its age is reset, extending the life of the bookmark in memory.

WARNING: Never attempt to access a bookmark directly in bookmark/cardmark memory space. Bookmark space is compacted from time to time and bookmarks may be relocated without your knowledge.

Example

```
extern struct IOSTdReq *IOReq; /* IO request opened earlier */

char Buffer[] = "Save this string";

IOReq->io_Command = CMD_WRITE;
IOReq->io_Offset = 0;
IOReq->io_Length = strlen(Buffer) + 1; /* string length with terminator */
IOReq->io_Data = Buffer;
if (DoIO(IOReq)) ProcessError(IOReq);
printf("%d bytes written/n", IOReq->io_Actual);

IOReq->io_Command = CMD_WRITE;
IOReq->io_Offset = 5;
IOReq->io_Length = 4;
IOReq->io_Data = &Buffer[5]; /* write just "this" */
if (DoIO(IOReq)) ProcessError(IOReq);
printf("%d bytes written/n", IOReq->io_Actual);
```

Errors

NOMARK	no bookmark exists
PASTEND	attempt write more than possible
BADARG	bad I/O argument

Related

CMD_READ
BD_CREATE

CMD_UPDATE

Reset the age of a bookmark.

Inputs

```
io_Command= CMD_UPDATE
io_Offset = 0
io_Length = 0
io_Data   = 0
```

Outputs

```
ErrorCode = io_Error
```

Description

Reset the age of a bookmark, thus extending its life in memory.

The age of a bookmark is used to determine its priority for being replaced by other bookmarks requiring memory. When a bookmark reaches a certain age, it becomes a candidate for removal so its memory can be reused. When its age is reset, it is renewed for another full lifespan.

The aging priority of a bookmark is initially set by the BD_Create command, and it may be changed with the BD_SETPRI command.

The CMD_READ and CMD_WRITE commands automatically reset the age of a bookmark.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command = CMD_UPDATE;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
if (DoIO(IOReq)) ProcessError(IOReq);
    printf("Updated/n");
```

Errors

NOMARK no bookmark exists

Related

BD_CREATE
BD_SETPRI

CMD_CLEAR

Clear the contents of a bookmark.

Inputs

```
io_Command= CMD_CLEAR
io_Offset = 0
io_Length = 0
io_Data   = 0
```

Outputs

```
ErrorCode = io_Error
```

Description

Clear the entire contents of a bookmark to zero, but do not delete the bookmark.

This command is only needed if you wish to erase the contents of a bookmark, but not free its memory.

When a bookmark is created (BD_CREATE), it is cleared automatically.

The age of the bookmark is reset.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
IOReq->io_Command = CMD_CLEAR;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
if (DoIO(IOReq)) ProcessError(IOReq);
printf("Cleared/n");
```

Errors

NOMARK no bookmark exists

Related

BD_WRITE

BD_TPEMEM

Return the type code of a particular module of the bookmark/cardmark memory space. (Not for applications!)

Inputs

```
io_Command= BD_TPEMEM
io_Offset = 0
io_Length = 0
io_Data   = MemoryRegion
```

Outputs

```
TypeCode = io_Actual
```

Description

This is an internal command used primarily by memory card manufacturers, vertical market developers, and snoopy application developers. It is used to help configure blank memory cards.

The code returned in `io_Actual` determines the current use and state of the memory. The values are:

>0 when the memory is already in use for bookmarks/cardmarks

=0 when the memory is available for use

<0 when the memory is not available (being used for diagnostics, expansion RAM, ROM, RAMDisk, etc.)

This command has no relationship or connection with memory controlled by the Exec library. It should not be used with general system memory.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command = BD_TYPEMEM;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = MemoryRegion;
DoIO(IOReq);

if (IOReq->io_Actual > 0)
    printf("Bookmark/Cardmark memory/n");
else
    if (IOReq->io_Actual == 0)
        printf("Available memory/n");
    else
        printf("In use by something else/n");
```

Errors

None

Related

BD_SIZEMEM

BD_INITMEM

BD_CREATEDDEV

BD_SIZEMEM

Determine the size of a particular module of bookmark/cardmark memory. (Not for applications!)

Inputs

```
io_Command= BD_SIZEMEM
io_Offset = 0
io_Length = MaxSize
io_Data   = MemoryRegion
```

Outputs

```
ByteSize = io_Actual
```

Description

This is an internal command which attempts to determine the size of a region of memory available for bookmarks. This command is normally executed after BD_TYPEMEM indicates that the memory is available for use.

This is a destructive memory test. It will alter the contents of the memory region. It should not be performed on active bookmark/cardmark or system memory.

Memory will be scanned to a resolution of 2K. The scan will be stopped when:

- The memory cannot hold a test value.
- The memory wraps over itself.
- The maximum size is reached as specified in io_Length.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
IOReq->io_Command = BD_SIZEMEM;
IOReq->io_Offset = 0;
IOReq->io_Length = 256 * 1024;
IOReq->io_Data = MemoryRegion;
Do*IO(IOReq);

printf("%d bytes in size/n", IOReq->io_Actual);
```

Errors

None

Related

BD_TYPEMEM
BD_INITMEM
BD_CREATEDEV

BD_INITMEM

Initialize a particular region of bookmark/cardmark memory. (Not for applications!)

Inputs

```
io_Command= BD_INITMEM
io_Offset = 0
io_Length = MemorySize
io_Data = MemoryRegion
```

Outputs

None

Description

This is an internal command which will initialize a region of memory for use as bookmarks/cardmarks. Its primary purpose is to "format" new memory cards. It clears the memory, sets up the memory header, and determines the maximum size of bookmarks for the memory.

This command is normally executed after **BD_TYPEMEM** and **BD_SIZEMEM** have been performed.

The **io_Length** parameter limits the amount of space to be used for storage of bookmarks (and the header). This allows you to reserve space in the memory region for other uses.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command = BD_INITMEM;
IOReq->io_Offset = 0;
IOReq->io_Length = MemorySize;
IOReq->io_Data = MemoryRegion;
DoIO(IOReq);
```

Errors

None

Related

BD_TYPEMEM

BD_SIZEMEM

BD_CREATEDEV

BD_CREATEDEV

Put a new bookmark/cardmark device on-line. (Not for applications!)

Inputs

```
io_Command= BD_CREATEDEV;
io_Offset = 0;
io_Length = DeviceName;
io_Data   = MemoryRegion;
```

Outputs

```
DevBase = io_Actual;
```

Description

This is an internal command used to create a new bookmark/cardmark device and put it on-line. It can be used to configure and install new memory cards. It will create and initialize an Exec device node and its function pointers, then add the device to Exec so that it may be accessed with **OpenDevice()**.

The device name is passed as an argument. For cardmark devices, this name should be "cardmark.device". Be sure to allocate this name someplace where it will not be freed when your program exits.

This command does not initialize the memory region being installed. You must use **BD_INITMEM** before executing this command.

The device base address of the functioning device is returned in **io_Actual**. If a **NULL** is returned, the **MakeLibrary()** function failed (out of memory).

Example

```
Extern struct IOStdReq *IOReq; /* IO request opened earlier */
#define MARKNAME "cardmark.device"
char *name;

IOReq->io_Command = BD_SIZEMEM;
IOReq->io_Offset = 0;
IOReq->io_Length = 256 * 1024;
IOReq->io_Data = MemoryRegion;
DoIO(IOReq);

size = IOReq->io_Actual;

IOReq->io_Command = BD_INITMEM;
IOReq->io_Offset = 0;
IOReq->io_Length = size;
IOReq->io_Data = MemoryRegion;
DoIO(IOReq);

name = AllocMem(Strlen(MARKNAME)+1,0);
if (!name)
    BadNews();
else
    strcpy(name,MARKNAME);

IOReq->io_Command = BD_CREATEDEV;
IOReq->io_Offset = 0;
IOReq->io_Length = (LONG) name;
IOReq->io_Data = MemoryRegion;
DoIO(IOReq);

if (!IOReq->io_Actual)
    BadNews();
```

Errors

None

Related

BD_TPEMEM

BD_SIZEMEM

BD_INITMEM

BD_CREATE

Create a new bookmark/cardmark entry.

Inputs

```
io_Command= BD_CREATE
io_Offset = BookmarkId
io_Length = MaxBytes
io_Data   = 0 and ln_Pri = Pri
```

Outputs

```
ErrorCode = io_Error
```

Description

This command is used to create new bookmarks. This must be done before a CMD_WRITE command. A bookmark of the requested size will be allocated from the memory associated with the

device you opened in the call to `OpenDevice()`. The `io_Offset` field must contain a valid bookmark identifier consisting of both a manufacturer id and product code. If the bookmark already exists, you will receive an error, and the command will not be performed.

The maximum size of a bookmark is restricted to allow many bookmarks to share memory.

The priority of a bookmark comes from the `ln_Pri` field of the I/O Request node. This value establishes the initial age of the bookmark. It may range between -128 and +127. You will need to determine the importance of your bookmark relative to other bookmarks.

The contents of a new bookmark are cleared to zero.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

char Buffer [] = "new bookmark";

IOReq->io_Command = BD_CREATE;
IOReq->io_Offset = MY_BID;          /* Must be valid Manuf/Prod Id */
IOReq->io_Length = 64;
IOReq->io_Data = 0;
IOReq->io_Node.ln_Pri = 0;
if (DoIO(IOReq)) ProcessError(IOReq);

IOReq->io_Command = CMD_WRITE;
IOReq->io_Offset = 0;
IOReq->io_Length = strlen(Buffer) + 1 /* string length with terminator */
IOReq->io_Data = Buffer;
if (DoIO(IOReq))
    ProcessError(IOReq);
```

Errors

TOOBIG	size is bigger than allowed
EXISTS	bookmark already exists with this id
NOSPACE	out of bookmark memory

Related

BD_DELETE
CMD_UPDATE
CMD_WRITE

BD_DELETE

Delete a bookmark/cardmark entry.

Inputs

```
io_Command = BD_DELETE;
io_Offset = 0;
io_Length = 0;
io_Data = 0;
```

Outputs

```
ErrorCode = io_Error
```

Description

This command deletes a bookmark and frees its memory.

Attempting to delete a non-existent bookmark will return an error.

If you attempt to access this bookmark once it has been deleted, you will receive an error.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
IOReq->io_Command = BD_DELETE;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
if (DoIO(IOReq))
    ProcessError(IOReq);
```

Errors

NOMARK no bookmark exists

Related

BD_CREATE

BD_MAXSIZE

Determine the maximum size allowed for a bookmark/cardmark.

Inputs

```
io_Command= BD_MAXSIZE
io_Offset = 0
io_Length = 0
io_Data = 0
```

Outputs

```
ByteSize = io_Actual
```

Description

This command returns the maximum size permitted for a bookmark. The value returned is dependent on the device you specified in the call to `OpenDevice()`. For example, the maximum size for cardmarks is normally larger than that of bookmarks due to the larger memory size. Also, memory cards may vary in size.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
IOReq->io_Command = BD_MAXSIZE;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
DoIO(IOReq);

printf("%d byte maximum/n", IOReq->io_Actual);
```

Errors

None

Related**BD_CREATE****BD_AVAIL****BD_AVAIL**

Return the amount of free space available in bookmark/cardmark memory.

Inputs

```
io_Command= BD_AVAIL
io_Offset = 0
io_Length = 0
io_Data   = 0
```

Outputs

```
ByteSize = io_Actual
```

Description

This command returns the total number of bytes available in bookmark or cardmark memory. The value returned is dependent on the device you specified in the call to `OpenDevice()`.

Although a considerable amount of space may be available, applications are limited to a maximum bookmark size to allow room for other applications.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command = BD_AVAIL;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
DoIO(IOReq);

printf("%d bytes available/n", IOReq-io_Actual);
```

Errors

None

Related**BD_MAXSIZE****BD_SIZEOF**

Return the size of a bookmark/cardmark.

Inputs

```
io_Command= BD_SIZEOF
io_Offset = 0
io_Length = 0
io_Data   = 0
```

Outputs

```
ErrorCode = io_Error
ByteSize = io_Actual
```

Description

This command returns the number of types allocated to a bookmark for data storage. It does not include system structures associated with the bookmark. The value is the same as that specified in the originating BD_CREATE.

The size of a bookmark is not affected by reading, writing, or clearing it. The size cannot be changed without deleting and recreating a new bookmark.

If no bookmark exists, this command returns an error.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command = BD_SIZEOF;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
if (!DoIO(IOReq)) ProcessError(IOReq);
    printf("%d bytes in size/n", IOReq->io_Actual);
```

Errors

NOMARK no bookmark exists

Related

BD_CREATE
BD_MAXSIZE

BD_SETPRI

Change the aging priority of a bookmark.

Inputs

```
io_Command= BD_SETPRI
io_Offset = NewPri
io_Length = 0
io_Data = 0
```

Outputs

```
ErrorCode = io_Error
OldPri = io_Actual
```

Description

This command changes the priority of a bookmark and returns the previous value. The higher the priority of a bookmark, the longer it will remain when memory has been exhausted and garbage collection has started.

Priorities can range between -128 and +127. The normal value is zero. Use zero if you are not sure.

The age of the bookmark is reset to a new value depending upon the new priority. (See CMD_UPDATE.)

Before changing the priority of a bookmark, determine its importance relative to other bookmarks (see previous chapter).

The initial priority of a bookmark comes from the `ln_Pri` field of the I/O request node used for `BD_CREATE`. Its value establishes the initial age of the bookmark.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command = BD_SETPRI;
IOReq->io_Offset = -10;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
if (!DoIO(IOReq)) ProcessError(IOReq);
    printf("Old priority was %d/n", IOReq->io_Actual);
```

Errors

NOMARK no bookmark exists

Related

BD_CREATE

BD_UPDATE

BD_CHECK

Calculate the checksum for the entire bookmark/cardmark memory space.

Inputs

```
io_Command = BD_CHECK
io_Offset = SetFlag
io_Length = 0
io_Data = 0
```

Outputs

```
ErrorCode = io_Error
Checksum = io_Actual
```

Description

This command calculates the checksum of all words within the bookmark/cardmark memory space. It will return zero if nothing in memory has been modified since the last checksum command; otherwise it will return a new checksum value.

The new checksum result will not be updated internally unless the `io_Offset` is set TRUE (non-zero).

The primary purpose of this command is to support application debugging efforts. This command can be called at various points in a program to determine if the bookmark/cardmark memory is being corrupted by bad indirection pointers.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
```

```
IOReq->io_Command = BD_CHECK;
IOReq->io_Offset = TRUE; /* Save checksum internally */
IOReq->io_Length = 0;
IOReq->io_Data = 0;
DoIO(IOReq);

printf("Checksum: %d/n", IOReq->io_Actual);

IOReq->io_Command = BD_CHECK;
IOReq->io_Offset = FALSE;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
DoIO(IOReq);

if (IOReq->io_Actual)
    printf("Checksum different: %d/n", IOReq->io_Actual);
```

Errors

None

Related

None

BD_PURGE

Purge the entire bookmark/cardmark memory. (Not for applications!)

Inputs

```
io_Command= BD_PURGE
io_Offset = 0
io_Length = 0
io_Data = 0
```

Outputs

Description

This command erases the entire bookmark memory including all system headers. It is supplied for testing purposes (for use with BD_DUMP, BD_LOAD) and for special memory card vendors. This command should not be used in normal applications.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */

IOReq->io_Command + BD_PURGE;
IOReq->io_Offset = 0;
IOReq->io_Length = 0;
IOReq->io_Data = 0;
DoIO(IOReq);

printf("Checksum: %d/n", IOReq->io_Actual);
```

Errors

None

Related

BD_INITMEM

BD_DUMP

BD_LOAD

BD_DUMP

Dump the bookmark/cardmark memory to main memory. (Not for applications!)

Inputs

```
io_Command= BD_DUMP
io_Offset = ByteOffset
io_Length = NumberOfBytes
io_Data   = Buffer
```

Outputs

```
BytesMoved = io_Actual
```

Description

This command provides a means duplicating bookmark/cardmark memories. With it you can write a program that will dump the contents of your bookmark/cardmark memory to a disk file for later loading back with the BD_LOAD command. This is often helpful during application testing.

The `io_Offset` field can be used when the memory must be buffered in smaller chunks to conserve memory space. When using this technique, the `io_Actual` field can be checked to determine when the entire memory has been transferred.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
extern long file;

#define BUF_SIZE (16 * 1024)
char Buffer[BUF_SIZE];
long offset = 0;

do
{
    IOReq->io_Command = BD_DUMP;
    IOReq->io_Offset = offset;
    IOReq->io_Length = BUF_SIZE;
    IOReq->io_Data = Buffer;
    DoIO(IOReq);

    Write(file,Buffer,IOReq->io_Actual); /* should check for file error */
    offset += BUF_SIZE;
} while (IOReq->io_Actual == BUF_SIZE);
```

Errors

None

Related

BD_LOAD

BD_LOAD

Load the bookmark/cardmark memory from main memory. (Not for applications!)

Inputs

```
io_Command= BD_LOAD
io_Offset = ByteOffset
io_Length = NumberOfBytes
io_Data   = Buffer
```

Outputs

```
BytesMoved = io_Actual
```

Description

This command provides a means of restoring previously dumped bookmark/cardmark memories. With it you can write a program that will load the contents of your bookmark/cardmark memory from a disk file. This is often helpful during application testing.

This command can also be used to write memory cards as part of a mass production duplication process.

The `io_Offset` field can be used when the memory must be buffered in smaller chunks to conserve memory space.

Example

```
extern struct IOStdReq *IOReq; /* IO request opened earlier */
extern long file;

#define BUF_SIZE (16 * 1024)
char Buffer[BUF_SIZE];
long offset = 0;
long size;

while ((size = Read(file, Buffer, BUF_SIZE) > 0)
{
    IOReq->io_Command = BD_LOAD;
    IOReq->io_Offset = offset;
    IOReq->io_Length = size;
    IOReq->io_Data = Buffer;
    DoIO(IOReq);

    offset += BUF_SIZE;
}
```

Errors

None

Related

BD_DUMP

CDTV Printer Preferences

One of the things that makes the CDTV system stand out from similar interactive consumer CD-ROM machines is that even the base unit has I/O ports to connect to external hardware like modems and printers. CDTV applications need to support printing wherever appropriate to push this advantage home. The CDTV printer preferences library is designed to allow easy user selection of printer settings as well as easy application access to those settings.

There were two problems involved in providing access to the Amiga printer drivers from a CDTV application. The first was to determine where and how the user's printer settings are stored. The non-volatile RAM (NVR) seems an obvious choice for storage and it is a good one; the question of storage format was not so obvious. It would be advantageous to retain the standard Amiga V1.3 Preferences structure to access the printer settings, especially because there is an instance of the struct Preferences attached to the *printer.driver* itself. Unless we want to make a custom CDTV printer driver, we need to retain compatibility with struct Preferences. On the other hand, we also want to fit into a single NVR division. The solution was to create a *prtprefs.library*, which is responsible for loading and saving the printer preferences to and from the NVR.

The second problem arises when you consider the question of how the user is going to change the printer preferences settings. The V1.3 Preferences editor is not especially usable from the CDTV remote control. In addition, it does not visually fit into the rest of the CDTV system, and contains many more settings than the average user wants to deal with. So a special CDTV Printer Preferences program was created which matches the Player Preferences program in appearance, and keeps the number of choices the user can make to a minimum. The CDTV printer preferences program also avoids the use of words to make localization simple.

You're On Your Own If You Need More. Some graphics programs may require providing the full range of printer preferences choices to the user. In those cases, the application will have to bring up its own "advanced printer options" screen, and use the Intuition `SetPrefs()` or `UsePrtPrefs()` function to communicate this information to the *printer.device*.

Using The Printer Preferences Library.

Using the printer preferences library is much like using any other disk based library. The *prtprefs.library* resides on your application disk, in the LIBS: directory, and is opened by making a call to the `OpenLibrary()` Exec function with "*prtprefs.library*" and 0 as the arguments.

```
#include <exec/types.h>
#include <exec/libraries.h>
#include <cdtv/cdtptrtprefs.h>

struct Library *PrtPrefsBase

void (main(void))
{
    if ((PrtPrefsBase = OpenLibrary("prtprefs.library",0)) = 0)
```

```

    {
        printf("Error: Can't open prtprefs.library\n");
        exit(20);
    }
    ... printerprefs code goes here
CloseLibrary(PrtPrefsBase);
}

```

The global variable `PrtPrefsBase` is used internally for all future library references until the library is closed by the application.

This library contains code for to read the NVR into the system Preferences structure, to save the printer portion of the system Preferences structure into the NVR, to set the System Preferences according to the current settings, and to bring up the CDTV printer preferences editor.

The commands in this library are:

ReadPrtPrefs()

This command is used to read and uncompress the CDTV Printer Preferences stored in the NVR into a standard Intuition Preferences structure. This routine will return an error status if the user has never set the CDTV printer Preferences. In that case, the application should call `AskCDTVPrtPrefs()` to bring up the Preferences program to give the user an opportunity to make the appropriate settings for printing.

SavePrtPrefs()

This command is used to compress and store the printer section of a standard Amiga Intuition Preferences structure into the NVR.

UsePrtPrefs()

This command is used to set the current printer Preferences by sending the information to Intuition. This will be later passed on to the *printer.device*.

AskUserPrtPrefs()

This command brings up the *CDTVPrinterPreferences* program, which allows the user to select and modify most of the preference settings relating to the *printer.device*. The *CDTVPrinterPrefs* program will take about 50K when in operation. If there is not enough memory to bring up the *PrinterPrefs* screen this call will fail.

Calling AmigaDOS. This library makes AmigaDOS calls. It should be called from a process rather than a task.

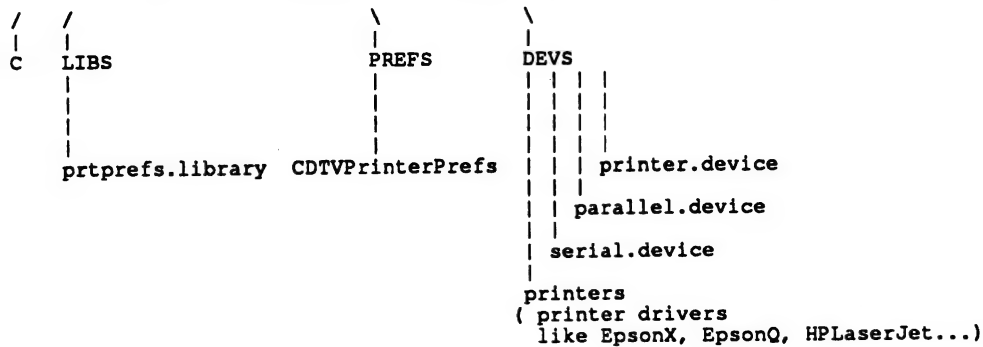
Setting Up Your Application For Printing.

The *prtprefs.library* must be placed in the directory `CD0:LIBS` of your application disc so that it may be found by the `OpenLibrary()` system call.

The *CDTVPrinterPrefs* file must be placed in the directory `CD0:PREFS` of your application. This file is executed whenever the `AskUserPrtPrefs()` call is made by your application. In addition, the *Run* command must be available in the `C:` directory.

The *printer.device*, *parallel.device*, and *serial.device* must be present in the `CD0:DEVS` directory. In addition, there must be a subdirectory called `PRINTERS` in the `DEVS` directory. `PRINTERS` is

Basically, you need to create an arrangement on your disc as follows:



Once all the files are properly placed on the application disc, your application will have access to the printer.

Using The Printer Device From A CDTV Application.

Adding printing capability to a CDTV application takes some planning. While using the *printer.device* for text and graphics output is straightforward and well documented in the Amiga ROM Kernel Reference Manuals (RKMs), to fit printing *naturally* into a CDTV application may require some effort.

The first decision is the mechanism for the user to initiate printing. In general, adding a "PRINT" icon to each of the screens is the most practical way. Another possibility is to reserve one of the keys of the remote for printing; this make sense only if printing can be initiated at any time. If printing is only enabled on certain screens, making the print operation a hot key function will lead to user frustration.

The second decision is whether to print in text mode or perform graphics dumps. Naturally if pictures or images are to be output, graphic dumps are the only possibility. But if it is just a text entry that is going to be printed, either type of printing can be used. The benefit of the graphic dump is that the printed page will look like the page displayed on the CDTV, format, font and all. The drawbacks of using a graphic dump are that it takes longer to send that much data to the printer, and that a graphics dump can take more memory than a text dump. (Memory seems to be a frequent concern for CDTV applications.) On the other hand, text sent to the printer must be formatted to appear at least somewhat similar to the screen display. This may require the application to contain a separate formatting system just for the printer.

Depending on the application (and the preference of the user) it may be better to print the entire article, rather than just the current screen. In that case there is the additional factor of the extra memory that would be required to image the entire article for a graphics dump. The “strip printing” method documented in the printer device chapter of RKM’s may almost be a necessity, due to memory limits. Text printing has some advantages in that case.

When printing from a CDTV application, AmigaDOS requesters should be turned off by setting the Process pr_WindowPtr to -1. This will turn off requesters like "Printer Trouble" and "Out of

The autodocs for the functions in the *prtprefs.library* follow.

TABLE OF CONTENTS

```
prtprefs.library/AskUserPrtPrefs
prtprefs.library/ReadPrtPrefs
prtprefs.library/SavePrtPrefs
prtprefs.library/UsePrtPref
```

prtprefs.library/AskUserPrtPrefs

prtprefs.library/AskUserPrtPrefs

NAME

AskUserPrtPrefs—display the printer Preferences to the user.

SYNOPSIS

LONG AskUserPrfPrefs(struct Preferences *p)

PARAMETERS

p A pointer to a Preferences structure containing printer preferences to be examined and possibly edited by the user.

RETURNS

One of the following values:

AU_SAVE 'Save' and 'Use' the result preferences.

AU_USE 'Use' but do not 'Save' the result preferences.

AU_CANCEL No change in preferences ('Use' them but 'Save' only if they haven't been saved before—see `ReadPrtPrefs()`).

AU_BUSY Routine is busy (another task is currently displaying the preferences for the user).

AU_LOWMEM There is not enough memory available to create the prefs display.

AU_NOTFOUND The prefs executable file could not be found on the disc.

DESCRIPTION

This routine will invoke the CDTV printer preferences screen editor to ask the user to inspect the preferences that are set in the supplied Preferences structure, 'p'. Since a graphics display is used for this editing a sufficient amount of Chip ram must be available. The user is presented with a sub-set of the entire preferences set for his/her inspection. They will exit by selecting an equivalent of one of the standard "Use", "Save" or "Cancel" buttons. The return code will indicate which was selected:

AU_SAVE means "activate the resultant prefs *and* save them permanently". Upon this selection, the application program would normally call `UsePrtPrefs()` *and* `SavePrtPrefs()`.

AU_USE means "activate the resultant prefs but *do not* save them permanently". Upon this selection, the application program would normally call `UsePrtPrefs()` *only*.

AU_CANCEL implies that the user does not want to edit the preferences, but would like to keep the prefs that were in effect when the editor was first invoked. Upon this selection, the application program should call `UsePrtPrefs()` if it hasn't already, and `SavePrtPrefs()` only if the preferences have not been previously saved (`ReadPrtPrefs()` returned FALSE). When the user selects "Cancel", `AskUserPrtPrefs()` restores the supplied preferences before returning; therefore 'p' is always correct after this call.

Negative return codes indicate an error. Only one call to this routine can be active at any given time, from all the Tasks in the system. An error is returned if it is busy (a non-blocking semaphore is used to prevent multiple callers). **AU_BUSY** indicates that another task is already editing the prefs. **AU_LOWMEM** indicates that there was insufficient memory available to create the prefs display – free some and try again. **AU_NOTFOUND** indicates the that editor executable could not be located on the disc. It could also mean the the AmigaDOS RUN command wasn't available for the `Execute()` AmigaDOS operation under 1.3.

prtprefs.library/ReadPrtPrefs

prtprefs.library/ReadPrtPrefs

NAME

ReadPrtPrefs—get the printer Preferences from NVR.

SYNOPSIS

```

BOOL ReadPrtPrefs(struct Preferences *p)
DO                      AO

```

PARAMETERS

p pointer to the Preferences structure to receive prefs info.

RETURNS

TRUE if the CDTV printer preferences have been previously set
 FALSE if they have not

DESCRIPTION

This routine will attempt to read CDTV printer preferences from the NVR. The prefs are stored in the supplied struct Preferences. A TRUE return indicates that the printer preferences have been set, and therefore the returned information came from the NVR. A return of FALSE indicates that printer preferences have not been set, and therefore a set of default information has been returned. Note that the default information comes from the Intuition preferences,

which ultimately comes from the "devs:system-configuration" file. This allows the default printer preferences to be defined and setup by a developer at the time the CD-ROM is created.

This routine returns valid data in the struct Preferences fields which do not apply to the printer or serial port. Application programs are expected to preserve this extra information if they pass this structure back to any other routine in the prtprfs.library.

NOTE

When running on an Amiga or when the "bookmark.device" is not available this routine will always return the default preferences (from Intuition) and FALSE.

prtprfs.library/SavePrtPrefs prtprfs.library/SavePrtPrefs

NAME

SavePrtPrefs—Save the Preferences settings to NVR.

SYNOPSIS

```
LONG SavePrtPrefs(struct Preferences *p)
DO                      A0
```

PARAMETERS

P A pointer to a Preferences structure containing the printer preferences to be saved to the NVR

RETURNS

0, indicating no error, or the error code from the 'bookmark.device' as returned by DoIO() (see 'bookmark.h' for possible error codes).

DESCRIPTION

This routine attempts to save the supplied printer preferences to the NVR. It will create the bookmark if it does not already exist. The error code that is returned could have resulted from the BD_CREATE call or the CMD_WRITE call, and it should be interpreted in this context. This routine *does not* activate the saved preferences; use UsePrtPrefs() to do that.

NOTE

-111 is a special return code which indicates that a signal bit could not be allocated for I/O with the bookmark.device.

This routine is non-functional on an Amiga or when the "bookmark.device" is not available – it will return an appropriate error in this case: BDERR_OPENFAIL (-1).

prtprfs.library/UsePrtPrefs prtprfs.library/UsePrtPrefs NAME

UsePrtPrefs—Use the printer Preferences.

SYNOPSIS

```
void UsePrtPrefs(struct Preferences *p)
A0
```

PARAMETERS

p

A pointer to a Preferences structure containing the printer preferences to be placed into use.

RETURNS

None

DESCRIPTION

This routine will activate a set of CDTV printer preferences by passing the information along appropriately to Intuition. This routine *does not* save the prefs to the NVR; use `SavePrtPrefs()` to do that.

NOTE

This routine expects valid data even in the struct Preferences fields which do not apply to the printer or serial port (this is returned by `ReadPrtPrefs()`).



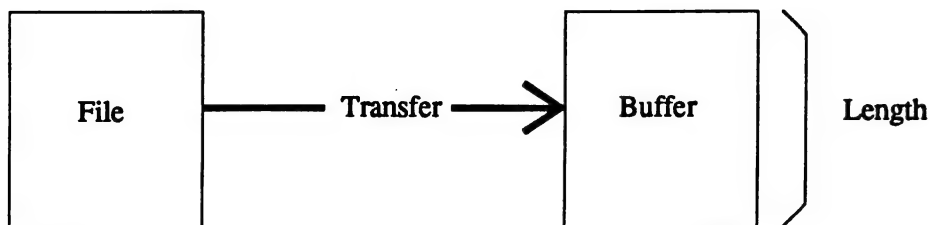
The Power of CDXL

CDXL is the name of Commodore's exclusive data transfer technology which permits applications to load data from a storage device at the fastest possible speeds. CDXL is not a filesystem or a compression/decompression system, nor is it limited to just 'video'. It is a new way of thinking about data transfer.

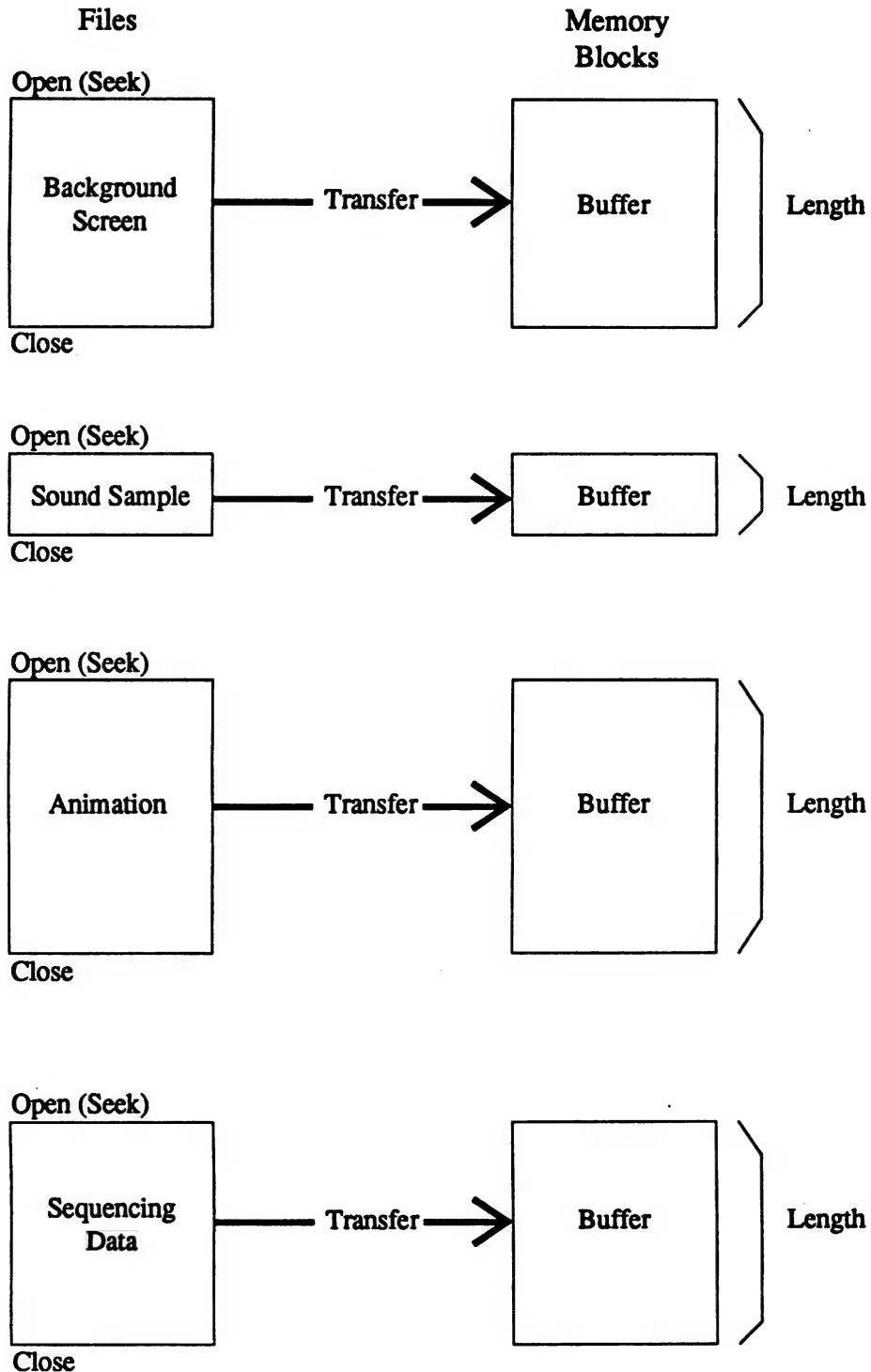
A File In Name Only. Although the term "file" is used throughout this article, it is used for conceptual simplicity. In actuality, CDXL is not part of the filesystem, but a low-level device driver request that operates on the raw sequential sectors of the storage device.

The Old Slow Way

Traditionally, reading data from a storage device is done through a function call or through a device I/O request. Further, each request fills only a single buffer of a specified length:



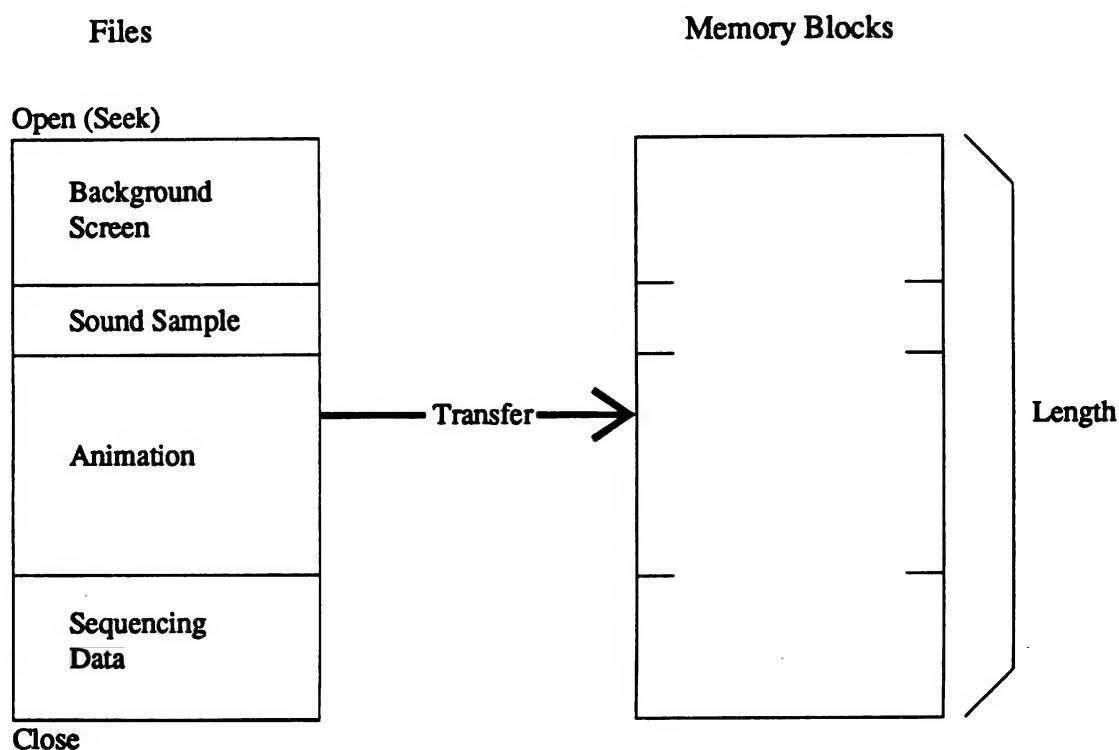
Further, many applications keep separate "objects" in separate files (the title screen is separate from the sound sample is separate from the animation, etc.). While this simplifies bookkeeping and project organization, it forces the application to access these files as separate objects. The inefficiency of this approach is particularly evident if the objects are related in some fashion:



Access to separate files on a storage device usually involves a seek when moving from file to file. On a hard drive, the time consumed by seeking is small enough to be considered insignificant. On a CD-ROM drive, however, seeks are extremely slow (up to four seconds on some inexpensive drives) and impact heavily on CD-ROM-based applications. Thus, the above seek-intensive model is inadequate for CD-ROM applications.

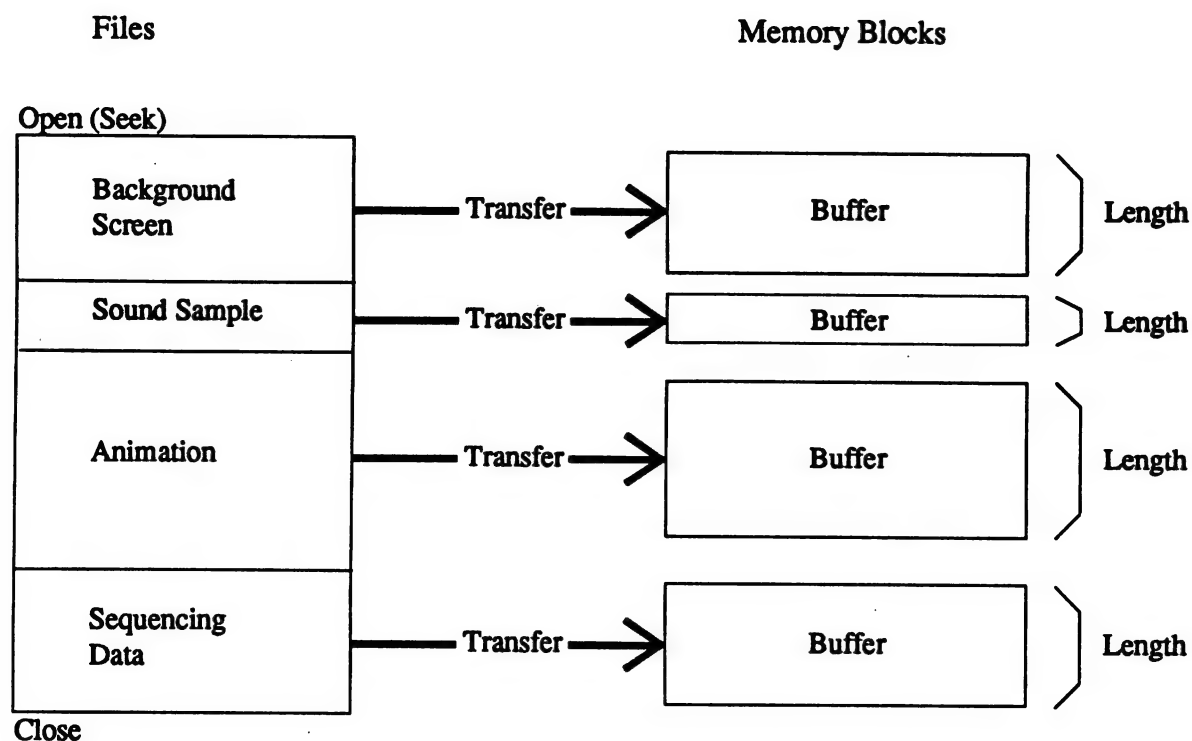
Improving The Model

Clearly, a primary key to improving performance on a CD-ROM platform is to minimize or eliminate seeks. A good way to do this would be to collect related data objects together in a single file. This way, the entire contents of the file can be loaded in a single operation:



The primary drawback to this approach is that it is rarely practical or even possible to arrange for the different data types to occupy contiguous areas in memory. It is usually necessary to have the different data occupying distinct memory areas (this is particularly true of graphics intended for display).

It is therefore necessary to modify the above model. While the data is still collected in a single file, we no longer perform a single read, but rather a series of reads into separate buffers:



In this model, the data is still in a single file, and seeks are minimized. However, for the above example, there are still four distinct transfer operations. When the read for each buffer is completed, what happens? Data movement has stopped. You've filled a buffer, but the disk is still turning, and there are buffers yet to be filled.

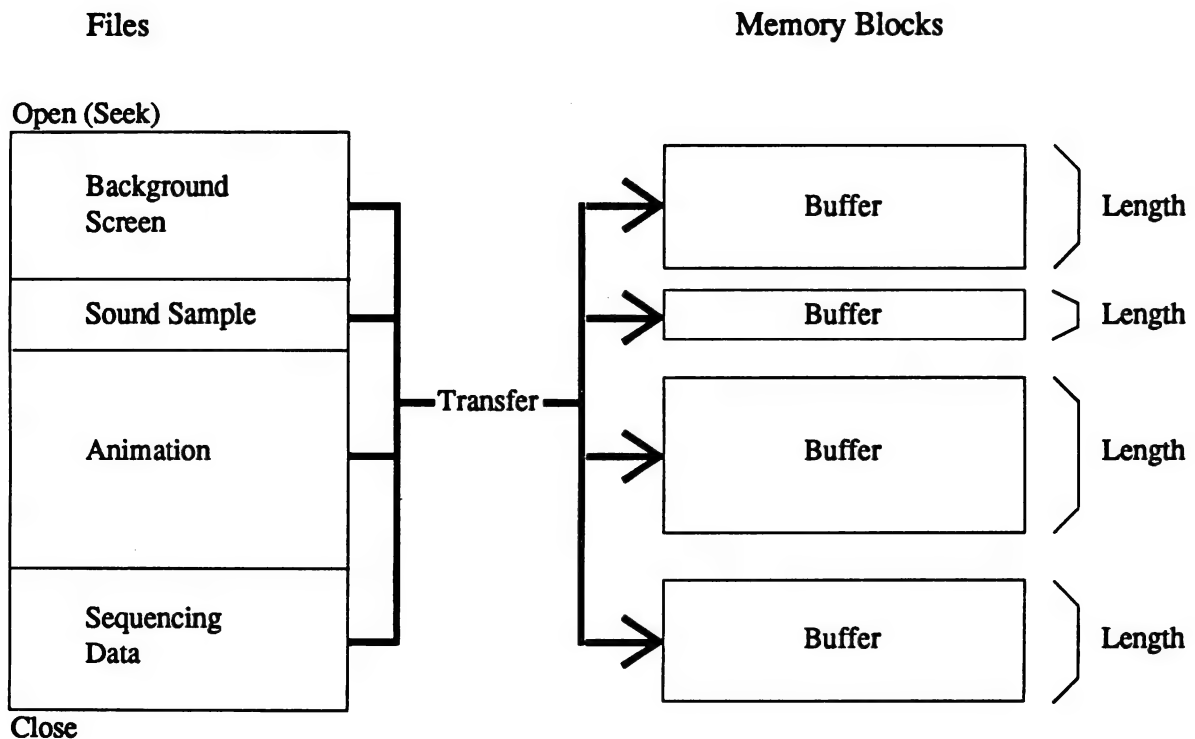
If your system is sufficiently quick, you can issue a new read command and perhaps catch the next sector before it spins out from under the head. However, a great deal of internal set up for the new transfer needs to be done, which strictly speaking shouldn't be necessary, since you're reading sequential bytes off the disk. But the device driver can't possibly know that you're going to eventually request additional bytes, and besides it can't know where to put them until you tell it. You could post an asynchronous request in advance in the hopes that the driver is smart enough to do an optimal operation. Unfortunately, most device drivers aren't this clever, and there's no guarantee that some other task won't get in and issue its own request in the middle of yours. Also, there is some overhead involved in the mere act of issuing so many requests.

So in all likelihood, you'll miss the sector and have to wait for it to spin around again. On a CD-ROM system, rotational latency can be significant; since CDss have a spiral track, missing a sector requires a re-seek to recover it.

Enter CDXL

CDXL is the next logical step in data transfer methodology. CDXL directly supports the “single file, multiple buffers” model outlined above. It takes advantage of the knowledge that the data you wish to read is laid out contiguously on the storage device. Using this fact, it sets up the device for a single large transfer, and automatically switches between your buffers as they are filled. In this way, the maximum transfer rate off the device is achieved.

All this complexity is handled within the device driver. You prepare a description of the buffers you want filled in the form of a linked list and submit it to the driver as a single request. CDXL then fills your buffers in one operation:



CDXL also has facilities in the form of call-back vectors to inform your application when given buffers have been filled. For example, you may wish to know when the background screen buffer has been filled so that you may display it. You can specify a call-back vector to be invoked when the transfer into the buffer is complete. This routine could send a signal to your application informing it that it is now okay to display the buffer.

Beyond The Obvious

With some imagination, CDXL can be used to do more than just a one-shot load of your buffers. Since CDXL takes a linked list of buffer descriptions, each of which can have a call-back vector, most of these possibilities lie in the skillful manipulation of the CDXL lists.

One popular example is to create a circular list describing two display buffers. When passed to the driver, CDXL will endlessly alternate between them. If a call-back vector is set up on each CDXL list node, a short routine can signal the application to display the newly-filled buffer. In this way, extremely long animations may be played off the disk at the device's maximum possible transfer rate.

Note that CDXL imposes no restrictions on the size or shape of the display buffers; that decision is left entirely up to you. All CDXL does is move bytes where you tell it. Thus, you could have the "traditional" 1/3 screen rectangle with a 4:3 aspect ratio, or you could have a tall overscanned vertical strip, or a wide horizontal strip, or a really small rectangle so you can increase the frame rate.

CDXL also allows you to manipulate the lists from within the call-back vector. From here, endless possibilities open up with regard to title design, flexibility, and on-disk data organization. For example, based on the contents of a standard header, you could dynamically switch the data over to any one of several buffers.

Make It Happen

Through its sheer simplicity, CDXL offers unparalleled speed and flexibility to the CDTV developer. In combination with existing software technologies and the power of the Amiga, CDTV presents the multimedia developer with the fewest obstacles to title creation and offers capabilities unavailable on any other platform.

CDXL Toolkit

Overview

This package contains the first generation of CDXL tools to help you edit video/audio sequences and write programs to play the results. The distribution disks contain:

- Eight separate tool programs.
- A standard C include file.
- An example source file.
- Example files in CDXL format.

Before using the tool programs, be certain to take the precautions stated in the next section.

The CDXL tools provided are:

XLMake

Creates or appends to a CDXL file. This tool accepts one or more IFF files (in the desired resolution and color mode), converts them to the standard CDXL format, and joins them together in sequence. Normally you would run this program as part of your video building script.

XLPlay

Simulates the playback of a CDXL file with approximate CDTV timing. XLPlay provides a simple means of viewing and hearing CDXL sequences on a standard Amiga computer as if they were on CDTV.

XLInfo

Displays detailed information about a CDXL file. The primary purpose of this tool is to provide size and timing information.

XLJoin

Combines multiple CDXL files into a single sequence. It can also insert a CDXL sequence into a second file at a particular point.

XLCopy

Copies frames from any point in a CDXL file and transfers them to a new file.

XLTrim

Removes frames from a CDXL file. Frames can be removed from any part of the file, or a file can be shortened overall by a specified amount.

XLAudio

Inserts an audio track into a CDXL file, i.e., Sound on Film. The audio will become an integral part of the CDXL sequence and will be heard when you run XLPlay.

XLClean

Rebuilds the frame sequence numbers and linkage information within a CDXL file. This action is not normally required, but is sometimes useful if custom CDXL programs have been used.

Each of these tools will be described in detail shortly.

Warning—Read This

CDXL sequences can be massive. For each minute of CDXL, nine megabytes of disk space are required. The CDXL Toolkit programs perform operations that are very read/write intensive. Should your Amiga crash (power failure, etc.) during an important operation, your AmigaDOS disk structures may become invalid and require reformatting.

Special precautions must be taken. It is well advised that you edit CDXL files on a separate disk drive, or at the very least, a separate partition of your drive. This has two advantages:

Safety

If your system crashes during CDXL editing, you may lose some of your work in the disk/partition, but the rest of your system and code will not be affected. If you keep your application source code on the same system, this is a very important consideration.

Performance

After a few days/weeks of CDXL editing, your disk/partition will become fragmented. Such fragmentation will introduce seek delays into your editing operations, which will cause the tools to slow down, and may result in poor timing and glitches in the XLPlay command.

So these are good arguments for taking the time to get set up properly. Believe us, it will be worth the effort! Partition your disk or use another disk for editing!

On our single disk systems, what we normally do is: put the standard workbench, libraries, devices, etc., into a small (6 meg) partition; put our development tools and code into a second (40 meg) partition; and use the rest of the disk for CDXL editing.

Commands

The CDXL Toolkit consists of a number of tools that operate as Shell (CLI) commands (they are not useful from the Workbench). Each of the tools requires command arguments appropriate to its function. A brief summary of the arguments can be obtained by executing the command without any arguments.

Generally, each tool accepts both an input and an output file as arguments. The tool performs its function by reading the input file a frame at a time, making the required changes, and then writing to the output file. Normally, the input file is not modified (*XLAudio* has a special exception). This approach helps preserve your original files from accidental modification, but at the cost of extra disk space.

Note that the performance of the tools will vary depending on the speed of your Amiga processor and hard disk. The primary bottleneck, however, will usually be the AmigaDOS File System—large CDXL files may require significant time to process.

All of the tools deal with files that use a standard type of file format and header. Files that do not contain the proper header cannot be processed. This header is described later.

XLMake

XLMake is the primary tool for creating CDXL sequences. The purpose of this tool is to accept standard IFF files that are already in the desired resolution, color mode and depth, translate them into a raw bitplane format, and join them together as a sequence of frames to be read with CDXL. Each frame contains header information about its size, resolution, colors, and audio.

This tool is normally used in a script file or ARexx program during the capture of individual video frames; for example, you might use it along with the *Art Department Professional* (from ASDG). Each time a new frame has been captured, scaled, and color adapted, *XLMake* is invoked to combine the frame to the CDXL file under construction. To help illustrate its use, a useful AREXX script has been provided on the distribution disks.

For this tool to run you must have *iffparse.library* installed on your system (in LIBS:). (A copy of it is provided on the distribution disk for those who are unfamiliar with this library).

Summary

```
XLMake [options] <IFFFile1> [<IFFFile2>...] <File>
```

The IFF files are translated and appended to the File. If the File does not exist, it will be created. If it does exist, new frames are appended to the end.

Options

-a<n>

Allocate space for audio track. The value specified is the number of bytes to use per frame. It is not necessary to allocate the audio during *XLMake*. You can always add the audio later with *XLAudio* (see *XLAudio* for more information). This feature is included in *XLMake* so you can view playback with accurate timing before your audio has been added.

-d

Forces AVM (Advanced Video Mode) and DCTV. If you are storing AVM pictures, this flag must be set.

-f<n>

Frame sequence number (not strictly required, but handy to know). Normally you would be assigning each frame number for every frame added, however, frame numbers are not required and *XLClean* can be used to renumber files later. When multiple IFF files are specified, the frame number is incremented automatically.

-h

Forces HAM mode (only needed for ancient Amiga programs that didn't set the CAMG correctly).

Example

```
XLMake -f10 -d intro.xl
```

```
XLMake -a1024 intro2.xl
```

XLPlay

XLPlay is a very useful program that simulates the playback of a CDXL file with approximate CDTV timing. *XLPlay* provides a simple means of viewing and hearing CDXL sequences on a standard Amiga computer as if they were on CDTV. The purpose of this tool is to allow developers to view their CDXL files in order to make decisions and changes without remastering a CD (or creating an emulator version).

Playback timing is only approximate. The general timing characteristics are preserved (total play time), but the speed of AmigaDOS and the lack of CDXL on standard Amigas prevents precise timing from being achieved.

If an audio track is present, it will be played along with the video (mono only). Slight audio "drop-outs" will be heard due to the varying lengths of disk read times in the Amiga File System.

Summary

`XLPlay [options] <File>`

The specified File is played on the Amiga screen. The image will always be centered in the middle of the screen.

You can stop the play before it has finished by pressing the Esc key. To pause/unpause press the Space bar.

Options

-h

Force hi-res graphics mode (not normally required).

-i

Force interlace graphics mode (not normally required)

-r<n>

Repeat playback a number of times. This is helpful for short sequences that would be difficult to see.

-q

Quiet (mute the audio).

-s[<n>]

Show frame sequence numbers during playback. Handy if you need to know where to cut and edit. The optional numeric value indicates the "brightness" of the frame number display. It may range from 0 to 15, where 0 is black and 15 is white.

Example

`XLPlay -r4 -s intro.xl`

XLInfo

XLInfo displays detailed information about a CDXL file. The primary purpose of this tool is to provide accurate timing and other information for application developers.

This tool will calculate and display the total play time, number of frames, frame sizes, frame resolutions, number of colors, color modes, audio track size, audio period estimate, sequencing information, and out of sequence frame numbers.

Normally only the first frame's header is examined, and an estimate is calculated from its size and the total file size. This is done to save time. To check the entire file, use the -w option.

Summary

```
XLInfo [options] <File>
```

CDXL file information and statistics will be displayed for the specified File.

Options

-a[<n>]

Estimate audio sizes. This will produce a table of what various audio track sizes would produce in terms of timing, playback period, total audio size, etc. If the optional value is provided, it will be interpreted as the number of bytes per frame for audio, and its relevant information will be shown.

-f<n>

First frame to examine,

-l<n>

Last frame to examine

The above two options use the ranges to examine only a section of a XL file. Information and statistics will only pertain to this section.

-w

Examine the whole file. Normally *XLInfo* just looks at the first frame and estimates the rest. If you have a mixture of frame sizes or wish to analyze the entire file, use this option.

Example

```
XLInfo -w intro.xl
```

XLJoin

XLJoin combines multiple CDXL files into a single file. The purpose of this tool is to provide a means of splicing together separate video scenes to create a single CDXL video file.

XLJoin also allows the insertion of CDXL files anywhere within a CDXL file. This tool does not alter its input files in any way, thus preserving them for use in other video sequences.

Summary

```
XLJoin [options] <FileIn1> [<FileIn2>...] <FileOut>
```

Each of the input files will be appended (in the specified order) and written to the output file. Frames in the output file will have new sequence numbers.

Options

-i<n>

Insert into the first file at given frame. Normally *XLJoin* just combines a list of XL files into one XL file. This option allows you to “splice” the other files into the first input file (still producing the “out” file). This is useful when you wish to add an XL file into another “master” XL file.

Example

```
XLJoin intro.xl action.xl end.xl all.xl
```

XLCopy

XLCopy copies frames from any point in a CDXL file and transfers them to a new file. The purpose of this tool is to provide a simple means of extracting useful video sequences without altering the original file. This approach is easier and safer than *XLTrim* in some situations.

Summary

```
XLCopy [options] <FileIn> <FileOut>
```

The specified portion of the input file will be copied to the output file. Frames in the output file will have new sequence numbers.

Options

-f<n>

First frame to copy (inclusive). May be specified alone or in conjunction with the -l or -n options. When specified alone (without an end frame or length), frames through the end of the file are copied.

-l<n>

Last frame to copy (inclusive). May be specified alone or in conjunction with the -f or -n options. When specified alone (without a start frame or length), frames from the start of the file are copied. Using -n for a frame count will copy that many frames up to and including the last frame specified.

-n<n>

Number of frames to copy. This can be used with -f or -l but never with both.

Example

```
XLCopy -f10 -n100 intro.xl newintro.xl
```

XLTrim

XLTrim supplies a simple set of editing functions for removing selected frames from CDXL files. It is useful for removing unwanted video or reducing the play time to some maximum value.

Options are provided to: trim video frames from the beginning, middle, or end; remove every Nth frame; and trim the entire file down to a specified number of frames or playback time.

This program also has the capability of removing the audio track from a CDXL file.

Summary

XLTrim [options] <FileIn> <FileOut>

The specified portion of the input file will be copied to the output file. Frames in the output file will have new sequence numbers.

Options

Options that specify frame numbers are inclusive.

- a**
Remove the audio track. Actually removes the track and frees that space from each frame.
- b<n>**
Remove from beginning to this frame.
- d<n>**
Delete every <n>th frame.
- e<n>**
Remove from the end back this number of frames. This option assumes that the file contains equal sized frames.
- f<n>**
First frame to start removing.
- l<n>**
Last frame to remove.
- n<n>**
Number of frames to remove.
- m<n>**
Maximum number of frames desired for entire file. Frames will be removed at evenly spaced intervals throughout the file in order to cut the file to this size. This option assumes that the file contains equal sized frames.
- t<n>**
Maximum play time in seconds. Frames will be removed at evenly spaced intervals throughout the file in order to cut the file to this size. This option assumes that the file contains equal sized frames.

Example

```
XLTrim -f10 -n10 intro.xl short.xl
    (removes frames 10-19)

XLTrim -m60 intro.xl intro60.xl
    (removes frames to make result 60 seconds)

XLTrim -e5 intro.xl introcut.xl
    (removes the last 5 frames)
```

XLAudio

XLAudio adds an audio sound track to an existing CDXL file. This operation is normally done as the last step in creating a CDXL audio/video sequence. The audio must already be adjusted to the correct sampling rate before laying down the track (see discussion below).

The tool will allocate the audio track for each CDXL frame if necessary, or overwrite the track if one already exists.

Only mono sound tracks are supported by this tool.

For this tool to run you must have *iffparse.library* installed on your system (in LIBS:). (The distribution disk has a copy of library for those who are unfamiliar with it).

Summary

```
XLAudio [options] <IFFSoundFile> <FileIn> [<FileOut>]
```

The IFF File is a sound file in either normal or compressed formats. FileIn holds the video sequence and may or may not already have an audio track. The output file contains both audio and video.

All frames in the input file must be of the same size.

Options

-a<n>

Allocate space for audio track. The value specified is the number of bytes to use per frame. See the discussion below to determine the correct size. Also, the *XLInfo* -a option will print a table of possible values to consider using.

-i

Insert audio into existing track. This action operates on the input file and no output file is required. The frames in the input file must already have space allocated for the audio track (from *XLMake* or a previous *XLAudio*). This option is useful when the XL file is extremely large.

-e

Ignore audio size errors when inserting. When inserting audio with the -i option, *XLAudio* assumes that the audio track will remain the same size. When it does not, *XLAudio* complains unless this option is set.

-p

Preserve audio compression. The IFF sound file is copied with its compression intact.

Example

```
XLAudio -a1288 audio.iff video.xl final.xl
```

XLClean

XLClean rebuilds the frame sequence numbers and linkage information within a CDXL file. This action is not normally required, but is sometimes useful if custom CDXL programs have been used.

Summary

```
XLClean <FileIn> <FileOut>
```

The input file is cleaned up and written to the output file.

Example

```
XLClean grab.xl video.xl
```

Creating Video

To a large extent, the techniques you use for creating video will depend on the equipment you have at your disposal. A wide range of possibilities exist depending on your budget, ingenuity, and patience.

Of course the simplest solution is to directly generate the video graphics on your Amiga by using your favorite 2D painting or 3D modeling/rendering program. The only requirement is that the program be capable of producing displayable IFF files as output. Each frame would be stored as a separate file (on hard disk or in RAM disk), and input to *XLMake* as needed.

Another approach might be to use a video frame digitizer that can be operated either manually or automatically (from *ARexx* or *ADPRO*). The images from this digitizer could be processed and altered with *ADPRO* then saved as an IFF file and again entered as input to *XLMake*.

An simple *ARexx* script to do this might be:

```
/* CDXL ADPRO program for PPS Framegrabber: */

OPTIONS RESULTS
ADDRESS "ADPro"
LFORMAT "FRAMEGRABBER"
SFORMAT "IFF"
GETNUMBER "How many frames?"

if RC != 0 then exit

FRAMES = ADPRO_RESULT

DO I = 1 to FRAMES
  ADDRESS "ADPro"
  OKAY2 "Ready to grab?"

  if RC = 0 then exit
  LOAD "XXX"
  SCREEN TYPE 0
  ABS SCALE 160 100
  EXECUTE

  SAVE "RAM:TempFrame" "SCREEN"
  ADPRO_DISPLAY
  PAUSE 100
  ADPRO_UNDISPLAY

  ADDRESS Command
  XLMAKE "RAM:TempFrame" FILENAME
END
```


Adding Audio

Suppose want to make a short video clip....

- 60 sec Total Run Time
- 14 K/sec audio quality (approx)
- 12168 bytes/frame (from XLInfo)

First determine what you know for sure.

Total Size:

$$60 \text{ sec} * 150 \text{ K/sec} * 1024 \text{ bytes/K} = 9216000 \text{ bytes}$$

Total Audio Size:

$$60 \text{ sec} * 14092 \text{ bytes/sec} = 845520 \text{ bytes}$$

Calculate basic figures.

CDXL less audio:

$$9216000 - 845520 = 8370480 \text{ bytes}$$

Number of frames:

$$8370480 \text{ bytes} / 12168 \text{ bytes/frame} = 688 \text{ frames}$$

Finally, calculate:

Audio sample size per frame:

$$845520 \text{ bytes} / 688 \text{ frames} = 1228 \text{ bytes / frame}$$

Now you can proceed as follows:

Use XLTrim to cut your grab down to 688 frames:

```
XLTrim -m688 video.grab video.688
```

With a program like *Audition4*, resample your audio to 14K:

Resample to 14092 bytes/sec

Save file as audio.14092

Finally, merge the video and audio data into a single file:

```
XLAudio -a1228 audio.14092 video.688 final.xl
```

Now, see how it looks and sounds:

```
XLPlay final.xl
```

Keep in mind that there may be a few small timing errors when playing this on an Amiga. Normally you won't see these, but you may hear them. Unfortunately, the only way to be absolutely sure is to test with the emulator using your application program.

Custom Programs

Within an application you will need code to play the CDXL sequences that you have created. You may also want to create special tools for manipulating CDXL sequences during the authoring process. To do this, you must understand the structure of a CDXL file and the philosophy behind it.

Standard CDXL files are made up of one or more frames. The general format of a frame is:

When a file contains more than one frame, the frames are simply concatenated, header and all. That is, one frame follows immediately after another. The only restriction is that frames must be aligned on even byte (word aligned) boundaries.

Various portions of a frame are optional. Actually, the only required section is the PAN Header, which contains information about the frame. The Color Map, Video Data, and Audio Data may or may not exist.

The PAN Structure is key to your access of CDXL files. It is defined in the file *pan.h*.

```
struct PanFrame
{
    UBYTE   Type;           /* type of structure being read */
    UBYTE   Info;           /* variations of type */
    ULONG   Size;           /* total size of frame */
    ULONG   Back;           /* offset from beginning of last frame */
    ULONG   Frame;          /* frame sequence number */
    UWORD   XSize;          /* width of video in pixels */
    UWORD   YSize;          /* height of video in lines */
    UBYTE   Reserved;       /* Hands off! MUST BE ZERO */
    UBYTE   PixelSize;      /* depth of pixel */
    UWORD   ColorMapSize;   /* # of bytes in color map */
    UWORD   AudioSize;      /* size of audio sample */
    UBYTE   PadBytes[8];    /* Hands off! MUST BE ZERO */
};
```

Type

This indicates what type of structure you are reading. Currently this field must be set to PAN_STANDARD.

Info

This field indicates variations on the general Type. Various options are discussed below.

Size

This is the total size of the frame in bytes. It includes the entire PanFrame structure, color map, video, and audio. With this value, you can skip to the next frame.

Back

This is the offset from the start of this frame back to the beginning of the previous frame. While CDXL cannot play backwards, this field is useful for tools.

Frame

This number uniquely identifies the frame within a file. A zero value indicates that the number is undefined, and tools print out a "?????" in the number field to show this.

XSize

The width of the video in pixels.

YSize

The height of the video in lines.

Reserved

Reserved for future. *Currently, must be zero!*

PixelSize

The "depth" of a pixel. That is, the number of bits used to indicate a color (with or without a color map). A zero value indicates that no video data exists within the frame. Example values: 4 for 16 color, 6 for HAM, 24 for RGB24, etc.

ColorMapSize

The total size of the color map in bytes. If you require more than 64K of color map, then you shouldn't be using CDXL (but you can use your own custom CDXL file, see below). A zero value indicates no color map. Typical values are 32 for 16 colors, 64 for 32 colors, 64 for half-bright (64 color) mode, 32 for HAM.

AudioSize

Total size of the audio sample area in bytes. A zero indicates no audio is present. Total size must include all channels (when multiple audio channels are used).

PadBytes[8]

This area is reserved for future use. For now it must always be set to zero.

The PAN Structure is optionally followed by the Color Map, Video data, and Audio data. Each of these areas must be even byte (16 bit word) aligned.

The **Type** field defined above is set to indicate the type of header structure being used. Over time, CDXL will evolve and this type will be critical for identifying what type of frame we are about to process. Right now the valid possibilities for this field are:

PAN_STANDARD

Used to indicate that the frame is in a standard format and that the above structure is used.

PAN_CUSTOM

Provided for you to create your own formats, but without causing problems for general tools. Custom formats allow you to lay your data however you need it. The only requirement is that the file start with the structure **PanHead**

```
struct PanHead
{
    UBYTE Type;
    UBYTE Info;
    ULONG Size;
};
```

Type

Set to PAN_CUSTOM to identify this as nonstandard.

Info

Reserved. Set to zero.

Size

This is the total size of the custom frame in bytes, including this structure. It is used to jump from the beginning of this frame to the beginning of the next frame.

Notes on The PAN Structure Fields—Type and Info

Type

PAN_SPECIAL

Reserved for experimental and proposed formats.

All other values for the Type field are reserved for new frame structures. As CDXL evolves, new PAN types will be published.

Info

In the PanFrame structure, the Info field describes various formats for the data contained within the frame.

PIV_MASK

Defines a bit field that tells you what video decoding should be used to display the image. Choices are:

PIV_STANDARD

For normal encoding (Amiga, RGB).

PIV_HAM

For hold and modify encoding.

PIV_YUV

For luma-chroma encoding (e.g., 844 bit YUV).

PIV_AVM

For CDTV advanced video mode and DCTV.

Commodore will add new values to this as needed. Pixel value orientation is *not* specified here. It is defined in a separate bit field, **PIF_MASK**, that indicates the orientation format:

PIF_PLANES

Data is interleaved on a bitplane basis, e.g., in the normal Amiga bitplane style. Each plane contains one bit of the pixel value.

PIF_PIXELS

Data is interleaved on per pixel basis, e.g., the chunky style like that used for VGA data, 8 bits per pixel, 24 bits per pixel, whatever.

PIF_LINES

Data is interleaved on a line basis. This can be thought of as each line being a single bitplane to itself.

Another field, **PIA_MASK**, indicates audio options. It tells you whether the audio is mono or stereo.

A few size macros are provided to help you move through a file. These should be used as needed.



CDXL Toolkit and Video Capture for CDTV

The CDXL Toolkit is now available to licensed CDTV developers. It provides an easy software solution to produce XL "video streams", files that contain video and audio sequences for display on CDTV systems.

Of course, the XL Toolkit is only the software end of the video production chain. It is designed to work in conjunction with video playback equipment, a framegrabber, and some extra software necessary for scaling and color balancing of images.

This paper discusses briefly the process, equipment, and software necessary to create a "video stream" for CDTV.

Real-time video capture

As described in the article in Appendix A, "Using ARexx And Videodiscs To Generate CDXL Files", to date there is no easy solution to transferring video sequences from tape to an XL stream on a hard disk. Ideally, you could capture video in real-time, direct to disk.

"Real-time video capture" systems are starting to ship on PC platforms, and have been advertised on the Macintosh®. Fluent Machines Inc. is shipping a single-slot AT-compatible board which performs real-time compression and decompression of full-motion digital video and audio data streams on a 386-based PC compatible. Their board, based on the ISO JPEG standard, is currently shipping, and costs \$3995 for the board plus \$995 for the developer software kit. For more information, contact Date Backer, Fluent Machines, (508)626-2144.

However, such systems are not currently available for Amigas. The price tag may also discourage potential CDXL authors. Thankfully, alternatives do exist.

Frame-by-Frame capture

Authors are currently producing CDXL streams by grabbing video one frame at a time, resizing the image, adjusting the palette, then using the CDXL toolkit to save the frame to an XL stream on disk. In order to digitize frame-by-frame, you need a video playback device capable of advancing across the video source frame by frame, and freezing the current frame, with minimal jitter, long enough to grab the frame, all under computer control.

We have tried video tape recorders, Hi-8 camcorders, and laser discs. We find the laser disc solution to be the most satisfying. See the discussion below for the advantages and inconvenience of each method.

ARexx scripts are used to control the entire video capture process. A sample ARexx script is attached.

Computer hardware

The basic computer hardware required to develop video streams is an Amiga 2500 or A3000, with minimum 3 Mbytes of RAM. A 68030-based system is required to obtain reasonable image processing speed.

A large SCSI hard disk is also necessary. It is suggested that the hard disk contain at least 2 partitions. The XL Toolkit reads and writes intensively to disk. It is strongly advised to avoid keeping source code or data on the same disk used for grabbing frames and creating XL streams.

Video Capture Hardware

A framegrabber is the essential requirement to enable video capture. The key feature is the ability to grab a frame in the highest possible resolution. Image processing software can be used later to drop down to the number of bitplanes required, and to scale the image down to 1/4 screen, if that is the target resolution. Use the highest possible number of colors when capturing the image to obtain optimal quality on your final frame.

If your video source is sufficiently stable, you may use slow-scan digitizers (such as *DigiView*) to grab the frames.

For the moment we have tested the following video capture devices:

Framegrabber, by Progressive Peripherals.

This framegrabber digitizes very rapidly; it is supported directly by The Art Department Professional. However, the image quality is only satisfactory—it does not grab in 24-bit mode. Progressive Peripherals ships this board in both NTSC and PAL flavors.

DCTV, by Digital Creations.

This system, recently introduced in PAL as well as NTSC, provides excellent quality images, using a minimal number of bitplanes. However, playback requires a CDTV equipped with the Advanced Video Board, or a DCTV module plugged into the back.

The following video capture boards, though not tested by the Commodore Special Projects group specifically for the purpose of CDXL video capture, should also work well:

Video Toaster, by NewTek.

This infamous board can be used for framegrabbing, although it is a bit slow for CDXL purposes. It saves in its own framestore format, which must be loaded into the *Toaster's* software, then saved to disk as a 24-bit IFF image. Image quality is excellent. NTSC-only, however.

Impact Vision 24, by Great Valley Products.

GVP's board works in 24 bits, in PAL or NTSC, but the video source must be RGB (not composite). An optional "RGB splitter" is required to grab incoming composite or S-VHS video.

In general, any framegrabber should work correctly. If possible, the software for the framegrabber should have an ARExx port, to enable automation of the grabbing process by an ARExx script.

Laser Disc Player

To date, the easiest way to capture frame by frame has been to press a videodisc of the source material. Videodiscs provide good video quality, maintain that quality even when displaying single frames, and are easily controlled via a computer. In the United States, a 30-minute videodisc can be pressed over a 5-day turn for about \$400. (The shorter the turn, the higher the cost.)

Playing back the videodisc with computer control can be performed by a number of players. Here is a list of the players supported by *AmigaVision* and by "Videodisc", an ARExx control program for video devices:

- Phillips 405
- Phillips 410
- Phillips 835
- Pioneer 2200
- Pioneer 4200
- Pioneer 6000
- Pioneer 6010
- Sony 1200
- Sony 1500
- Sony 1550
- Sony 2000

Video Tape Recorders

A good quality video tape deck (like the Sony Umatic series) is usually equipped with an interface for computer control. The *Videodisc* utility can control a Sony UMatic deck. Again, the cost of such systems eliminates them as an alternative for many developers.

Hi-8 Camcorders are generally less expensive than Umatic equipment. It is possible to control a Hi-8 camcorder equipped with an infrared control system, via the *Illumilink 2.0* package from Geodesic Publications. ((404)822-0566) This \$100 system lets an ARExx script on your Amiga control your Hi-8 Camcorder via its infrared receiver. Your script can advance to the next frame, pause the Hi-8 camcorder, grab the frame, and continue.

Watch Your Pause! Intensive use of the pause function on a Hi-8 camera may well damage the motor on the camcorder!

Image Processing Software

The *Videodisc* utility is an ARExx control system for video devices designed to work specifically with CDXL video capture systems. It is available from CATS. *Videodisc* lets ARExx scripts control video devices supported by the *Amiga player.device*. It represents a key part of the ARExx script used to create an XL video stream. *Videodisc* is described in Appendix B.

The *Art Department Professional* from ASDG Inc. is the second essential piece of software. *ADPro* includes numerous functions, all controllable via an ARExx script, to control framegrabbers, translate images from 24-bits to HAM, adjust to a particular palette, resize images, etc.

Audio Capture System

For CDXL streams, the audio sampling rate is usually fairly low (in the 8 to 15 Khz range) to maintain the highest possible number of video frames per second. Numerous 8-bit audio digitizers are available for the Amiga. We have used both *Perfect Sound*, *SoundMaster*, and *AMAS*. *Audio Master III* has been used to manipulate the digitized sounds.

A sample AREXX script from Videodisc vers. .12

```

/* Grab.rexx */
/* This demonstrates how to automatically build a CDXL file. */

/* The boss says "I wanna see RESULTS!" */
options results

/* Send commands to Art Department Professional */
address "ADPro"

/* We want to load frames from the FrameGrabber */
LFORMAT "FRAMEGRABBER"

/* We want normal IFF output files */
SFORMAT "IFF"

/* Search for the first frame of my scene */
address VIDEODISC.1 search 14173

/* We'll grab 1200 frames worth */
do frame = 1 to 1200

address "ADPro"
/* Grab the video */
load "it" "FIELD1"

/* I'm grabbing a letterboxed image, so I can throw some */
/* of it away */
OPERATOR "CROP IMAGE" 280 120 20 40
/* Shrink the image to the destination size */
ABS SCALE 192 82
/* Generate the HAM image */
EXECUTE

/* Save it where XLMake can get at it */
SAVE "RAM:Temp" "IMAGE"
/* Some commands for COMMAND */
ADDRESS COMMAND

/* Keep the user informed */
say "Saving frame #" frame
/* Append the frame we just saved onto the end of our CDXL file */
XLMAKE "-h" "RAM:Temp" "SDH0:bladerunner.xl"

/* I know I'm going to have 15 frames / second or less, */
/* so it's OK to only grab every other frame */
address videodisc.1 step 2

/* Keep doin' it 'til it's done. */
end

```

Appendix A—Using ARexx And Videodiscs To Generate CDXL Files

Introduction

One of the most difficult parts of using CDXL to display motion video has been the process of building a suitable CDXL file. There is currently no straightforward way to simply record video and audio material in real time directly into a CDXL file. In fact, there is currently no way (in real-time) to record video and audio into any type of file. Acquisition of the video material can be a tedious process where constant (and perhaps costly) human intervention can be required to advance the video material to the next frame.

While the new CDXL toolkit provides examples of how to use ARexx to automate most of the CDXL build process, the critical issue of controlling an external video device is not covered.

This article demonstrates the production of an actual CDXL file, and, in the process, demonstrates a new tool that can be used to speed up and automate the critical "next frame" step. The *Art Department Professional*, *ADPRO* is used throughout.

In an ideal world, making a CDXL file would simply be a matter of plugging your video player into the CDXL board, commanding the CDXL board to record the incoming video and audio to a CDXL file on disk, and pressing play on the video deck. Unfortunately, it's not that simple yet. Audio is easy; the video part is not so easy. For now, the best we can do is to grab a frame at a time.

Regrettably, single frame grabbing from video presents its own set of troubles. Most video is on videotape, but most video tape decks don't play single frames very well, and, of those that do, very few can be controlled by a computer. This leaves videodisc. Videodiscs have three advantages over videotape for creating CDXL files. First, they have exceptionally high video quality. Second, they can maintain that high quality even when displaying single frames. Third, it is relatively easy (especially now) to control videodisc players with computers.

This said, videodiscs do have one particular disadvantage—they generally cannot be recorded onto. Until recently, making a laser videodisc of a piece of video was relatively expensive, and took quite a while. Now, however, there are several companies that will make videodiscs for you at a reasonable cost. For \$400 (at most) and a five day wait, you can have your thirty minutes of video turned into a top-quality videodisc, ready for the next step in building a CDXL sequence.

Step-by-step

1—Gather video

Think about the places in your application that could benefit from motion video sequences. All of these scenes will need to be produced using normal video production techniques. Obviously, certain video elements do not lend themselves to CDXL presentation—things like overlaid text, low contrast dark images, or anything where having low resolution would prohibit the viewer from seeing important details.

2—Edit video

There are probably a lot of different scenes that you'd like to include in your finished application. Because videodiscs hold thirty minutes of information, and because the per-disc cost is the same no matter how much time is used, it is best to get as much video onto each disc as possible. If you have less than thirty minutes of video, consider filling out the time with video from some other project, or with outtakes—just in case. It's also important that your finished, edited tape meets the requirements of the mastering house that you choose. Most houses will want so many seconds of black, so many seconds of color bars, etc. Some may want the tape to have a continuous time code. Not having the tape set up the way the house wants it may mean paying a special "processing fee."

3—Mail it away

Relax and enjoy the five day wait.

4—The nitty gritty—Building the video portion of your CDXL files

It will be easier to follow this step if it's broken into substeps.

First, find the frame numbers of the beginning and end of your scene. Call them BEGIN and END.

Next, determine how long the scene should be in real time by computing $(\text{BEGIN}-\text{END})/30$. Call that number SECONDS.

Make sure that, when played (by pressing play on the disc player) the scene really is that long. (Don't laugh, if the disc was made from a film master, it may have 24 frame per second frame markers, which would cause the player to step through and search the video at 24 fps, even though it properly plays back at 30 fps. Making the wrong assumption here can lead to big timing problems later.)

Start *ADPro* and try grabbing a few frames of video. You'll want to try out a few, to get the color, contrast, and other settings just right. At this point, you should choose a palette for the scene. If you plan to put the CDXL image onto an existing screen, you'll want to load in that screen's palette. Once you got the palette you want, lock it.

Next, using *ADPro*, crop the picture down to the size that you want, and save the resulting image.

Now, use *XLMake* to build a one frame XL file that includes just the frame that you saved from *ADPro*.

Run *XLinfo* on that frame, and proceed with the calculations listed in the "Adding Audio" section of the "CDXL Toolkit" article.

At this point, you'll probably need to adjust the frame size of your images, in order to get the balance of frame-rate, image size, and audio quality that you want. Keep these two caveats concerning audio in mind.

- First, the Amiga has certain "natural" sampling and playback rates, which are based on the "period" value passed to the *audio.device*. Playing back samples at speeds other than the "naturals" is non-trivial. Most sampling software will not even let you sample at rates other than these "naturals." Unfortunately, *XLinfo* will calculate and report lots of in-between values for "Audio bytes/second."

- Second, while you may pass *XLInfo* an odd value for “audio bytes per frame,” *XLAudio* will only process an even number of bytes per frame.

The upshot of these two caveats is that you should try to get a balance of image size and frame rate that lets you have both an even number of audio bytes per frame and a “natural” value for audio bytes per second.

Now, using the sample script *grab.rx* as an example, build an ARexx script that steps through the video, grabs frames, processes them, and builds a CDXL file. If you know that the video is going to be running at 15 fps or less (12 fps for a film-sourced disc), then only grab every other frame. *Grab.rx* assumes the use of a *FrameGrabber*, but anything that can grab from videodisc will work. Since the videodisc image is a perfect still, you can even use slow-scan digitizers like *DCTV*, *DigiView* (with an electronic splitter), or *Deluxe View*.

5—Start 'er up and let 'er go.

There are a lot of variables which will affect the amount of time it takes to process one frame. Based on these variables, it can take anywhere from six to twenty seconds per frame to build a CDXL file. It can take quite a while. Of course, it's a lot faster than ray-tracing, so it's worth it.

6—Try it out.

At this point you'll probably be anxious to see the video part of your CDXL file. Just use *XLPlay*. One interesting note that may or may not have been fixed by the time you read this—If you run *XLPlay* while *ADPro* is running, your image may be way off the right-hand side of the screen. Go figure!? Just quit *ADPro* and try again.

7—Adding audio.

Once the video is in the form of a CDXL file, adding audio is simply a matter of following the instructions listed in the “Adding Audio” section of the “CDXL Toolkit” article. Again, keep the two “audio caveats” discussed earlier in mind when adding the audio.

Note that when you press play on most videodisc players, the audio begins immediately. This makes it easy to get the audio start sync'ed. Just use videodisc to search to the first frame of your scene, start your sampler, and press play on the player. (Of course you could also send the player a PLAY command.) Let the sampler record a little beyond the end of the scene, in case you need the slop. Remember that *XLAudio* will only use as much audio as it can fit into your CDXL file, so letting the audio file be a little longer than necessary can't hurt. When you've stopped the sampling, use the editing features of your sampling software to chop off the zeroes at the beginning of the sample, and you'll then have a sample that starts at exactly the first frame of your segment.

Also note that the *XLAudio* step can take a while. When *XLAudio* is working, it reads from two files and writes to a third. To minimize disk thrashing, consider putting the audio sample file in RAM: for this step. More importantly, it is highly recommended that you have the source and destination CDXL files on two different partitions (or drives.) That way, if something should go wrong while adding the audio, the original CDXL file (which probably took a long time to build) should still be intact.

8—That's all folks.

You should now have a finished CDXL file that can be played with *XLPlay*. You'll need your own code (which you can model after the *example.c* source in the CDXL toolkit) to actually play the CDXL file from within a CDTV application, but *XLPlay* will at least let you check that the finished file is ok. When playing with *XLPlay*, you may notice some audio timing errors. These should go away when playing from a CD (or the emulator.)

Conclusion

CDXL sets us apart. Use it. Use it well.

DRAW (Direct Read After Write) Videodisc Vendors

Technidisc, Inc.
2250 Meijer Drive
Troy, MI 48084-7111
313-435-7430
800-321-9610
fax: 313-435-8540

"SuperDisc - Exhibit Quality DRAWDISC"

Turnaround 1 Side CAV only Max. Quantity
5 day US\$400 3*
3 day \$600 2*
1 day \$750 1
Same day \$1000 1
*additional copies \$250

Optical Disc Corporation
(800) 350-3500
(213) 946-3050

Certified recording centers nationwide
RLV Disc service US\$300

Appendix B—Videodisc

A means of controlling videodisc equipment via ARexx

Introduction

The *Videodisc* program was designed to let ARexx scripts (and applications capable of sending ARexx messages) control any of the video devices supported by the Amiga *player.device*.

Player.device is an Amiga shared device that provides mid-level support for the playback of audio/visual materials from supported video disc players and video tape decks. It is commonly distributed with the *AmigaVision*TM authoring system.

When run, *Videodisc* opens an ARexx port, accepts straight-forward English commands, and translates them into the necessary *Player.device* commands.

Videodisc does not reply to your script or application until after it has completed an operation. For example, if told to play from frame 1 to frame 100, (using one command) your ARexx script won't continue until frame 100 has been reached. This prevents search latency from affecting the timing of a script or application.

Setup

Player.device should be in your DEVS: directory. This file is the actual shared device that accepts commands and sends them to another smaller program that is specific to each player. The file is included in the devs directory of the *Videodisc* distribution disk. It's also in the devs directory of the *AmigaVision* Boot Disk.

The directory *players*, and all the files in it, should be in your DEVS: directory. These files are the individual programs that accept mid-level commands from *player.device*, and convert them to player-specific commands to be sent out the designated port. Some video devices need different code for different baud rates, so there may be more than one file for a particular player.

The file *player-units* should be in your DEVS: directory. When *player.device* initializes, this file is read to determine which type of player should be used, which serial device should be used, which port number, and the baud rate to be used. See below for a sample *player-units* file. You configure this file to let *player.device* know what you're using.

Operation

You start *Videodisc* by simply RUNNING it from a CLI or shell. *Videodisc* will display an ARexx port name (usually VIDEODISC.1) and send an INITIALIZE command to the current video device.

When you wish to terminate *Videodisc*, send it a QUIT command. This can be accomplished from the command line by typing:

```
RX "address videodisc.1 quit"
```

ARexx Command Set

You send *Videodisc* commands by addressing the port named when *Videodisc* is run. Usually this will be "videodisc.1" If it's not, it means there is another copy of *Videodisc* running. Having two (or more) copies of *Videodisc* running at the same time does not provide any benefits. (You cannot, for example, drive two players simultaneously this way.)

The basic commands are listed below. Items in brackets are optional, items in parentheses are defaults, and items separated by slashes are mutually exclusive. Capitalized words are actual commands or keywords, lower case words describe arguments.

```
VIDEO [(ON) / OFF]
INDEX [CHAPTER / (FRAME)] [(ON) / OFF]
CX [(ON) / OFF]
STILL [frame number / (current frame)]
SEARCH [CHAPTER / FRAME] [number / (0)]
STEP [(FORWARD) / REVERSE] [number of frames / (1)]
WAITFOR [CHAPTER / (FRAME)] number
PLAY      [SLOW / (NORMAL) / FAST] [(FORWARD) / REVERSE]...
          [FROM [(FRAME) / CHAPTER] number]...
          [UNTIL [(TIME) / CHAPTER] number]
INITIALIZE
RESET
IDLE
QUIT
```

The valid ranges for standard arguments are as follows.

Frame numbers = 1-54000

Most players will go to the last frame on a disc if told to search to 54000, some will return an error, others will go to frame 1. Most players will go to frame 1 if told to go to frame 0, others will return an error, some do both.

chapter numbers = 0-99

Most players will go to the last chapter if told to search to 99, some will return an error.

Known Bugs

- Gets a lock on shell (shell window can't be closed).
- Cannot be stopped if a *Videodisc* command fails.
- Fixed parsing order on PLAY command.

Revision History

91.11.14	Version .12	Removed debugging info
91.11.04	Version .11	Added WAITFOR command
91.10.15	Version .1	First (sort of) working version.

Layout of devs:player-units file

The *player-units* file is a plain ASCII text file that is loaded and parsed when it *player.device* starts up. It is up to you to make sure that the file is properly configured for your setup.

Here's a sample file:

```
0 Sony_1200_9600 9600 serial.device 3
```

And an explanation of each field:

0

The "device number"—it will always be zero.

Sony_1200_9600

The filename of the "player file" that is to be used, in this case *Sony_1200_9600*. Note that you should not include the path, as these files must be in *devs:players*. Also note that some players have different files for different baud rates. Always use a file that is greater than or equal to the actual baud rate that you plan to use.

9600

Next comes the actual baud rate that the player is configured for, in this case, "9600". Note that most players have DIP switches which allow you to change the baud rate, but some have a fixed rate while others have an electronic menu that allows the rate to be changed.

serial.device

The name of a shared device that is to be used when communicating with the player. This should usually be *serial.device*. Do not give a path, as the file should be in *devs:*.

3

The unit number that is to be used with the *serial.device* (or whatever other device is being used.) This will usually be "0" as that is the unit number for the default port. In the sample, it is set to "3" because of using an A2232 multi-serial port board, and that's the unit number of the player.

These arguments must be in the stated order, separated by spaces, and the line should be terminated with a \n. (Or a press of the return key if you're editing it by hand.)



License Material Overview

CDTV developers are requested to sign a CDTV license agreement. This agreement gives you the rights subject to the terms of the agreement to do the following things in exchange for the royalty you pay to Commodore:

- Use the CDTV multimedia logo on your product and its advertising.
- Use the Mastering Software to create your master ISO 9660 image.
- Receive special materials available only to licensed CDTV developers.

What Does Commodore Do With The License Fees?

Commodore has introduced this licensing in order to ensure that CDTV discs can be easily identified as such by the consumer, offer uniformity in key user areas, and are of a high standard of quality and reliability. The revenue generated by the program will assist in CDTV system improvements including cost reduction, earlier licensing of the technology, and funding of additional development tools which will be made available to licensees.

How Are These Materials Obtained?

The entire set of licensed materials is sent to developers after Commodore has received a signed license agreement.

What Is Included In The Licensed Materials?

Commodore is regularly adding to the list of tools and materials available to licensed CDTV developers. The most recent addition was the CDXL Toolkit. We expect to make more utilities available in the future.

Here follows a list briefly describing the materials available to licensed developers.

CDXL Documentation

The CDXL documentation is a detailed description of the patent pending CDXL technology. This document provides background information on the technology, describes basic operation, and details the corresponding data structures.

CDXL Toolkit

This set of CLI-based utilities helps you capture and edit video/audio sequences and write programs to play the results. The disk contains:

- Eight separate tool programs
- A standard 'C' include file
- An example source file
- Example files in CDXL format

The CDXL tools provided are:

XLMake	Creates or appends to a CDXL file.
XLPlay	Simulates playback of a CDXL file from a hard disk drive.
XLInfo	Displays detailed information about a CDXL file.
XLJoin	Combines multiple CDXL files into a single sequence.
XLCopy	Copies frames from any point in a CDXL file into a new file.
XLTrim	Removes frames from a CDXL file.
XLAudio	Inserts an audio track into a CDXL file.
XLClean	Rebuilds the frame sequence numbers within a CDXL file.

ISO DevPak Diskette

This diskette contains the software necessary to build an ISO 9660 CD-ROM image from your AmigaDOS SCSI disk drive. Numerous utility programs are included, such as:

iso

Scans your AmigaDOS source diskette, and prepares a controlfile listing all the filenames and their directories, in ISO format.

buildtrack

Creates the ISO 9660 image. This image may then be sent to a pre-mastering center for creation of a write-once CD-ROM.

cdtv.tm

Contains the CDTV interactive multimedia logo, displayed on booting up all CDTV applications

rmtm

Removes the logo from the screen

bookit

Reads the CDTV preferences, to center the screen, set the Workbench colors to black, start the CDTV screen saver, etc.

Playerprefs Library

The *playerprefs.library* contains many useful routines allowing you to center a CDTV screen according to CDTV preferences, start the Audio control panel, read and update the CDTV preferences, and insert joystick inputs into the input food chain.

CDTV Tools Disk

This diskette contains a series of tools useful to developers. These include:

Audio Tools

Playtrack	A simple CD audio player, in source and executable.
NoReset	A utility which prevents the system reset when a CD is ejected.
Audio2	An 8SVX example

Include files

All the headers and include files necessary for CDTV development.

Compression

Debox	information on the compression routines included in the ROMs of the CDTV
SBox	A CLI-driven file compressor

Debug

A full set of Amiga debugging tools

Fonts

Helv 18 and 25 fonts, from Xiphias. These fonts, which are used in the Xiphias *TimeTable* products, are available to developers. These fonts may be included in your CDTV application, with no royalties payable.

Prefs

Utilities to read and write CDTV preference settings

CDXL

Sample source code in 'C' to set up and play back a CDXL transfer.



CDTV Graphics

CDTV is remarkable in part because it builds upon the innovation of the Amiga by combining it (in a very convenient form) with a mass storage device. The graphics abilities of the Amiga have always cried for such an addition and it is the CDTV user and programmer who will benefit. There are, however, caveats to be aware of. The unit will be used (almost exclusively) in ways in which home computers never have been. It is up to you, the CDTV application designer, to insure that your application is as useful and accessible to the CDTV user as possible. This article introduces CDTV's graphics abilities and draws attention to important considerations for application design.

Coprocessors

CDTV is able to process graphic information as quickly as it does because the CPU is aided by a number of coprocessor chips. One of these, Copper, controls nearly the entire graphics system. Among other duties, it controls register updating, positioning and rendering of sprites (graphic elements), modify color palettes, and control the blitter. The blitter (contained in the coprocessor chip known as Agnus) is capable of complex manipulation of large amounts of data. It is heavily used by the graphics system.

Information on programming the blitter can be found in the Addison-Wesley *Amiga ROM Kernel Reference Manuals*, but one important technical note is worth repeating here.

WARNING ON INTERMIXING BLITTER AND PROCESSOR MEMORY ACCESS or DEALLOCATION

Many of the *graphics.library* functions use the blitter, most notably those which render text and images, fill or pattern, draw lines or dots, and move blocks of graphic memory.

These functions generally operate in a loop, doing one plane at a time as follows:

```
OwnBlitter(); /* Get control of the blitter */
for (planes=0; planes < bitmap->depth; planes++)
{
    WaitBlit();
    start a blit
}
DisownBlitter(); /* Relinquish control of the blitter */
```

The code above always waits for the blitter at the start, and exits after the final blit has been started. It is important to note that when these blitter-using functions return to the caller, the final (or only) blit has been *started*, but not necessarily *completed*. If you are only intermixing such graphics calls, this is efficient, because the *next* graphics blitter call will wait for the blitter to be done *before* starting its first blit.

However, if you are mixing such graphics blitter calls with processor access of the same graphics memory, or if you plan to immediately deallocate or reuse any of the memory areas which were

passed to your last graphics blitter-using function call as a source, destination, or mask, then you *must* first `WaitBlit()` to make sure that the final blit of the graphics call is completed before you change or deallocate the memory it is accessing.

If you do not follow the above procedure, you will run into problems on faster machines or under other circumstances where the blitter is not as fast as the processor.

Graphic Modes

CDTV offers a wide array of screen modes (horizontal and vertical sizes) and depths (number of bitplanes, which determines the number of colors). Some are available immediately, under the 1.3 operating system. More will be available with the 2.0 operating system and the Advanced Video Mode hardware (discussed later).

Video Modes

The basic video modes and their attributes are charted below.

<u>Video Modes</u>	Default Resolution Horizontal x Vertical (in pixels)		<u>Needs 2.0?</u>	<u>Needs ECS?†</u>	<u>Maximum Colors</u>	<u>HAM/EHB‡ Available?</u>
	<u>NTSC</u>	<u>PAL</u>				
Lores	320x200	320x256	No	No	32 out of 4096	Yes
Lores-Interlaced	320x400	320x512	No	No	32 out of 4096	Yes
Hires	640x200	640x256	No	No	16 out of 4096	No
Hires-Interlaced	640x400	640x512	No	No	16 out of 4096	No
SuperHires	1280x200	1280x256	Yes	Yes	4 out of 64	No
SuperHires-Interlaced	1280x400	1280x512	Yes	Yes	4 out of 64	No

† The Enhanced Chip Set is a replacement set of graphics manipulation/display chips.

‡ HAM (Hold-and-Modify) and EHB (Extra-Half-Brite) modes are special graphic modes that build upon certain base modes and yield additional colors in a memory-frugal way.

HAM mode allows up to 4,096 colors on the screen at one time. In normal CDTV graphics modes as each value formed by the combination of bitplanes is selected, the data contained in the selected color register is loaded into the color output circuit for the pixel being written on the screen. Therefore, each pixel is colored by the contents of the selected color register.

In HAM mode, however, the value in the color output circuitry is retained, and one of the three components of the color (red, green, or blue) is modified by bits coming from two "control" bitplanes.

EHB mode allows twice as many colors as its base mode (up to 64 colors in either Lores mode) by giving each color in the base mode a companion color that is half as intense. For example, a base color of white (0xFFFF) would have the companion color grey (0x888).

See the Addison-Wesley *Amiga ROM Kernel Reference Manual* series of books for full and complete information on CDTV's graphics functions as well as descriptions of the CDTV's custom chips and graphics modes.

The article, "Getting the Best Image for Your CDTV Applications" (elsewhere in this chapter), contains a chart of the amount of computer memory required for the various screen modes.

CDTV Viewing

The CDTV user will (most likely) be viewing the application on a television rather than a computer monitor. Because of this, the CDTV developer must keep several important factors in mind when creating a CDTV title:

- Televisions are notorious for being adjusted to "overscan" the displayed image. This can result in objects rendered at the very edges of the display being "lost" behind the television tube's bezel. In addition, keeping screens of information centered (and keeping the most important graphics elements toward the center of the screen) are necessary to insure that the user doesn't miss anything of importance.
- The CDTV-to-television video connection will most likely be via RF (radio frequency) rather than direct video or RGB (each primary color separately) input. This is the least desirable method because of signal loss and excessive noise, but it is also the most commonly available method for inexpensive televisions.
- The NTSC (National Television Standards Committee) video signal does not handle high intensity values well. Avoid selecting colors which have RGB (Red, Green, Blue) values above 0xDDD (hexadecimal).
- The CDTV user will typically be seated across the room from the television screen. Text rendered on the screen must be large enough to be easily read. Ideally the user should be able to alter the size of the font used by an application if needed. Icons, selection buttons, etc. should also be sized such that they can be discerned from a reasonable distance.
- Televisions typically have coarser dot pitch (larger pixels) than a monitor. Diagonal lines may appear excessively jagged without some form of anti-aliasing. The user's distance from the television will actually help alleviate this problem.

The most important step the CDTV developer can take to prevent releasing an application which violates the above "rules" is to test the application on a "home" television throughout the development

Animation

If a picture is worth a thousand words, an animation must be worth millions. Thanks in part to its coprocessor chips, CDTV is able to display graphic images rapidly enough to simulate movement. There are powerful methods of compressing these images either to store more of them, read them in more rapidly, or both. Numerous software packages exist for the Amiga (and hence for CDTV) that can allow the artist and non-artist alike to create sequences of animation. Many of these packages include small "player" programs that can display the finished animation without the memory overhead of the package it was originally created with.

Commodore's proprietary CDXL (described elsewhere) can also be used to bring graphic information off of the CD-ROM rapidly enough to allow impressive near-video animation.

Differing Video Formats

CDTV is available in countries using the three major video transmission methods: NTSC (National Television Standards Committee), PAL (Phase Alternate Line) and SECAM (electronic color system with memory). It is important to remember that these different formats have different display specifications, and that an application designed under one of them may not look the same under a different format.

Chief among these differences is the number of scan lines. Examine the chart of basic video modes (above) to see the differences between NTSC and PAL. It is possible to query the user's CDTV to learn which type of video format it is configured for and account for the different format qualities.

Advanced Video Mode

The Advanced Video Mode (available shortly) will greatly enhance the color palette in the high resolution mode. A three bitplane AVM screen, which would normally yield only 8 out of 4,096 colors will instead give nearly 100,000. A four bitplane screen would similarly give nearly 4 million colors! Specific programming information will be available when the AVM is introduced to CDTV developers.

Getting The Best Image For Your CDTV Application

This article presents issues which developers should be aware of when generating images for use with the CDTV system. It will also present several techniques for maximizing image quality making use of ASDG's *The Art Department*.

General Issues For Displaying Imagery on Any Amiga

Following is a discussion of issues which relate to displaying imagery on any Amiga-based system.

Choice of Video Mode

The choice of video mode is one of the most basic decisions you will have to make when preparing an Amiga-based application. The choices which you must make relate to:

Resolution

The Amiga offers high and low resolution modes. Each offers some advantages but at some cost.

High resolution offers a sharper image and permits more information to be displayed at one time. However, it limits the number of colors displayable at one time and can degrade system performance.

Low resolution offers a richer color capability, can use less memory and does not affect system performance as much as high resolution. However, low resolution screens cannot display as much information and can appear "chunky."

Palette Depth

The number of colors which will be used during the display of images must also be considered. A richer set of colors nearly always produces more pleasing results. However, deeper palettes come at the expense of resolution (you must forego high resolution if you want more than 16 colors), memory, and animation speed.

Interlace

The vertical resolution of an image can be doubled by placing the Amiga into an interlaced video mode. Interlaced video does not increase the loading caused by the display upon the processor but it does double the memory requirement of any given display. The principal liability of interlaced video is the visible flickering that it can cause on machines which do not have a deinterlacer. Horizontal lines, especially when thin and highly contrasting to surrounding pixels, can produce an extremely noticeable flicker which can become annoying.

Overscan

By using overscan, the visible border around an Amiga screen becomes usable display area.

Overscan (which can be along the vertical and horizontal dimensions independently) adds to the TV-like appearance of an Amiga display. An overscanned screen consumes more memory and can drain a significant amount of processor speed especially in some high resolution screen modes. The degree of overscan which is necessary or acceptable varies depending upon whom you ask. Many people specify a maximum overscan screen as 768 pixels wide (in high resolution) by 484 high pixels (in interlace) in NTSC or 592 pixels high in PAL. However, a limit of 736 pixels wide in high resolution is recommended for best results.

Mixing Video Modes

On the Amiga, it is possible to mix different video modes on-screen simultaneously with the following restrictions:

- Different video modes may be horizontally stacked only. Two video modes cannot share the same machine.
- A small number of lines (approximately 3 non-interlaced lines) become unusable when transitioning from one video mode to another. Worse still, the mouse pointer will become obscured while passing through these lines.
- Transitioning from one video mode to another consumes a small amount of processor bandwidth.
- If any of the visible screen modes are interlaced, the entire screen will be displayed in interlace.

Within these restrictions, you can see that it is possible to create a display with (for example) text displayed in high resolution and four colors and an image displayed in low resolution with 64 colors.

Chip RAM Contention

Some Amiga display modes place a greater load on machine resources than others. The table below indicates the relative amount that a given display mode (without overscan) will affect the CPU when it attempts to gain access to a Chip memory location.

Horizontal Resolution	Number of Bitplanes					
	1	2	3	4	5	6
Low	None	None	None	None	Some	Moderate
High	None	None	Moderate	Considerable	—	—

Overscan can considerably decrease available processor time. When displaying a high resolution four bitplane screen, for example, the processor is locked out from accessing CHIP RAM during the display of an entire scanline. It can only access CHIP RAM during the horizontal and vertical retrace times. Horizontal overscan shortens the available horizontal retrace time and vertical overscan further diminishes the length of available vertical retrace time.

No Barriers To Fast. The processor is not impeded from accessing memory located on the Fast memory bus.

Memory Requirements

Each screen mode will consume a different amount of CHIP memory. The table below indicates how much memory is used (in bytes) for each of the given common screen formats:

Screen Size	Number of Bitplanes					
	1	2	3	4	5	6
320 by 200	8000	16000	24000	32000	40000	48000
320 by 400	16000	32000	48000	64000	80000	96000
640 by 200	16000	32000	48000	64000	—	—
640 by 400	32000	64000	96000	128000	—	—
368 by 240	11040	22080	33120	44160	55200	66240
736 by 480	44160	88320	132480	176640	—	—

Aspect Ratio

Each dot displayed on an Amiga screen has a specific shape. Unfortunately, this shape is not square. As a consequence, pixel aspect must be considered when displaying images from a wide variety of sources.

Many factors affect the aspect of on-screen pixels. These include:

Screen Format

Switching between high and low resolution doubles or halves the number of pixels displayed across the screen. Clearly, this affects the width of each pixel with high resolution pixels being half as wide as their low resolution counterparts. Similarly, switching between interlaced and non-interlaced video halves or doubles the number pixels shown vertically.

Dots on the Amiga screen most closely approach square when in:

- Low Resolution, Non-Interlaced
- High Resolution, Interlaced

In these modes, the ratio of a pixel's width to its height is approximately 10 to 11 in NTSC. Low resolution pixels shown in interlaced video are approximately 5 to 11 (width to height).

NTSC Or PAL

PAL video fits more horizontal lines onto the same sized display area. Therefore, PAL systems can come closer to square pixels than NTSC systems.

Monitor Settings

Every monitor has controls (often accessible by the user) which affect the width and height of the displayed image. Clearly, even the most careful planning and compensation for pixel aspect can be undone by the user.

Without considerations for pixel aspect, images scanned with most optical scanners (which produce square pixels) will appear stretched or distorted even when shown in lores/non-interlaced or hires/interlaced modes.

Other sources of square pixels include 3D modeling programs and images created on non-Amiga computer systems.

International Video Formats

As indicated in the previous section, the differences between NTSC and PAL will affect your product development by affecting the aspect of displayed pixels. NTSC and PAL also differ in how much displayable area is available on screen. The standard height of a non-interlaced, non-overscanned NTSC screen is 200 pixels. The same screen on a PAL system is 256 pixels high.

The problem that different imaging areas present is particularly nasty if you must have only one set of images for use on both NTSC and PAL systems. For example, if an image is created for proper viewing on an NTSC system, it will not fill as much of the screen and will appear distorted on a PAL machine. If an image is intended for proper viewing on a PAL system, then part of the image will be obscured on an NTSC system.

Therefore requiring only one set of imagery for use on both PAL and NTSC systems may be an unrealistic goal. The recommendation here is to create a separate set of images for use on PAL and NTSC systems. Since CD-ROMS are quite large, and other internationalization factors will come into play in your software anyway, this may not as unpleasant as it first appears.

CDTV Specific Issues

This section contains a discussion of issues which apply specifically to images for use on the CDTV system.

TVs Not Monitors

As a developer, you can expect the Amiga owner to have a high resolution computer monitor for use on his system. This is not the case with CDTV. The expected display device is an ordinary television. This affects you in two ways.

First, televisions vary incredibly in quality and sharpness. High resolution text (for example) which is perfectly readable on your development system may not be readable at all on a television.

Second, the interface to television, composite RGB, is not as precise as the analog RGB normally used in Amiga displays. High contrast transitions which appear perfectly clear on your development system may become ugly masses of bleeding color on a television.

Recommendations to overcome these problems include:

- No CDTV development environment is complete without a *cheap* color TV running in parallel with your high resolution computer monitor. The best way to anticipate how your product will look in the consumer's home is to view your product the same way the consumer will.
- Avoid high contrast transitions especially where text is concerned. Especially avoid saturated reds.
- Avoid images that are too bright. Specifically, try to keep your brightest colors at an intensity level of less than 13 (using the Amiga standard scale of 0 to 15).

Distance

The typical computer user views his high resolution computer monitor from no more than four feet away. The typical television viewer is generally 6 to 10 feet from the set. This means that visual detail may be lost simply because the user is further away from the display device than you had anticipated.

This can be a significant advantage, however, because:

- The advantages of dithering become even more pronounced as the viewing distance makes it unlikely that the user will be able to discern the individual dots.
- Low resolution displays (with their richer color palettes) can be much more effective than limited palette high resolution displays. The larger viewing distance means that richness of color will be much more important than sharpness of dots.

Phosphor Burnout

It is very likely that a CDTV user will leave the device on for long periods of time without actually being present to cause the screen display to change. If your application does not include a self contained "screen blanker", it is likely to burn a hole in the user's television set, not something which is likely to please your customer.

Getting The Best Image

This section describes various techniques which are helpful in getting the best image quality possible for your CDTV images. These techniques assume the use of ASDG's *The Art Department (TAD)* for image development.

Image Sources

It is possible that no personal computer has ever offered more alternatives for capturing or generating images as the Amiga. How you generate or capture an image can significantly affect the quality of your end product. Following is a summary of imagery sources and where we recommend their use.

Color Scanners

Color scanners should be used whenever you need to capture flat art. Video digitizers simply do not have the resolution to provide high quality imagery for a broad range of applications. Purely gray scale scanners are not recommended for CDTV applications since CDTV is primarily a color device. Additionally, color scanners can scan in gray scale and cost only marginally more. Color scanners also offer advantages in speed and ease of use compared to other color image capture systems.

Video Digitizers

Video digitizers are recommended for capturing non-flat art or for capturing images from a live or video source. The quality of the images you can capture with a video digitizer is strongly affected by the quality of your video source. Specifically, where a video camera is being used, the quality of the camera can make or break the image.

Paint Boxes or Paint Programs

There are a number of high-end paint boxes which can create enormously detailed imagery such as the *Quantel* and *Waverfront* systems. Also, images created with any personal computer based paint programs such as *Deluxe Paint*, *ColorRIX*, *TIPS*, or *RIO* can be employed.

3D Modeling Systems

3-D modeling systems can play an essential role in image development. This is especially true for the creation of complex scenes which cannot be scanned or digitized since they do not actually exist in the real world.

Dithering

Dithering is one of the most important techniques for increasing visual realism in your CDTV images. Dithering sacrifices some of the spatial sharpness of the image to dramatically increase the color fidelity of the image.

Carried to an extreme, dithering can produce the appearance of true gray scales on a single bitplane monochrome screen. More typically, the dithering techniques found in *TAD* can produce the impression of hundreds of colors on a 16 color screen, or the impression of many thousands of colors on an Extra-Half-Bright or Hold-And-Modify screen.

TAD offers seven dithering methods (including the choice of no dithering). We have found our Floyd-Steinberg implementation to be suitable in most instance, especially when creating images for display on high resolution screens.

Dithering cannot increase color fidelity where no increase is possible. For instance, there is no point in dithering a 32 color picture if the original image data contained only 32 colors. In other words, for dithering to have an effect, there must be more colors in the original data than there will be in the rendered image.

TAD can synthesize new colors, however, when its scaling function is used. For example, if a 640 by 400, 32 color image is scaled down to 320 by 200, *TAD* can synthesize as many as thousands of new colors as it performs the reduction. This is accomplished by pixel averaging all the different combinations of the original colors. In this way, the spatial resolution lost in reducing the size of an image can often be compensated for with increased color range.

Dithering can sometimes produce unwanted, seemingly stray dots. These can be eliminated within *TAD* by slightly increasing the contrast and rendering, or by invoking the RIP (Remove Insulated Pixels) functions.

Aspect Correction

TAD offers highly precise scaling both upwards and downwards in size. When correcting for aspect, don't forget that the width can be enlarged rather than always shrinking the height.

An easy way to determine just how far off-square your Amiga/monitor combination is, is to draw a 1 inch square on paper and then scan it in. Using *TAD*, display the alleged square in many diverse screen formats and experiment with the scaling functions to gain experience on making a square, square.

Interlace

Making An Interlaced Image

Interlaced low-resolution images, especially in HAM, can appear exceedingly crisp on a CDTV display. Interlaced low-resolution is, however, one of those resolution combinations which is nowhere near square in aspect.

To produce an interlaced low-resolution image, simply take your square or near-square aspect image and scale down the width by approximately fifty percent. Alternatively, you could scale up the height by one hundred percent.

When To Interlace

Natural images (people, places, etc.) can generally be displayed in interlace without significant flicker problems. Images which have a lot of horizontal lines or very stark transitions from color to color (such as some images created with 3D graphics programs) will fare very poorly when displayed in interlace.

Overcoming Interlace Flicker

If you have a problem image which you need to display in interlace, try the following:

- Try reducing the contrast of the image slightly. This may cause any flickering scan lines to become more subdued and therefore less noticeable.
- Try scaling the image upwards or downwards slightly. This may cause flickering scanlines to be spread over more or fewer displayed scanlines and therefore be less noticeable.

Increasing Visual Punch

Contrast

As indicated in the preceding section, a slight increase in contrast can eliminate seemingly stray dots when rendering a dithered image. A slight increase in contrast can also add a considerable amount of visual punch to an image.

Gamma Correction

TAD offers variable Gamma Correction (non-linear color correction) which allows you to brighten an image without loss of detail which the standard brightness control would cause. Increasing the Gamma value of an image can dramatically increase the visual punch of an image, can be used as a special effect, or can bring out detail in a dark image.

Who's Afraid of Gray Scale

Sixteen shades of gray (especially when dithered by *TAD*) covers the spectrum of grays far better than even 64 or 4096 colors covers the spectrum of color. Don't be afraid of using gray scale images in your application.

TAD offers a color to gray scale conversion function which takes into account the relative frequency response of the human eye. The color to gray conversion function can produce some exceedingly realistic images.

Flips And Mirrors

If your subject matter allows for flips and mirror images, you can increase your composition flexibility by using a flip or mirror of an image rather than the original image. For example, if a person looking off to the left simply looks better than a person looking off to the right, flip him.

Avoid flips or mirrors when text is visible in the image or when technical data or drawings when flipping would either be noticeable or change the meaning of the image.

Genlock Considerations

If it is possible that your application may be used on a CDTV on which there is a Genlock device. In this case, Genlocked video will appear through any areas in your imagery which are drawn in pen (or color register) 0.

You can easily make images Genlock-opaque in *TAD* by instructing *TAD* not to use color register 0 during its rendering. This is accomplished in the Palette control panel by selecting a non-zero color offset and requesting a CUST (or custom) rendering.

Mixing Computer Chosen And Manually Chosen Colors

If you wish to render text directly over an image you may wish to take the text colors into consideration when producing the palette for the image. If the text colors are not taken into account, you will be forced to use one of the colors appearing in the image. This often produces unacceptable results.

Using *TAD*'s palette controls, you can set aside color registers to be used for titling in two ways.

The first method produces images which do not have the reserved color register appearing anywhere in them. This can be done using the same technique as reserving color register zero described in the previous section. Simply decrease the number of colors to be used and set the offset of color zero to the desired value.

For example, to reserve four colors for titling within a 32 color image:

1. Set the total number of colors to 32. This defines the depth of the resulting image.
2. Set the number of colors to be used to be 28.

3. Set the offset of color zero to either 0 or 4. Setting this value to zero reserves a block of 4 colors after the 28 colors used by the image. Setting this value to 4 reserves 4 colors prior to the 28 used by the image.
4. Render the image using the CUST setting.
5. Set the 4 reserved colors to their desired values.
6. Save the image to disk.

This will produce an image which will not contain any reference to the unused block of registers. You can modify the unused registers to any value and not affect the look of the image.

The second method allows you to set aside a given number of colors as before, but then allows you to merge these reserved colors into the overall image. This allows you to specify specific colors which must be present in the image, and then lets *The Art Department* do the rest.

Jumping into the above sequence of steps at step 7:

7. Lock the palette so that TAD does not recreate new colors.
8. Set the offset of color zero to zero.
9. Set the number of colors to be used to the total number of colors available.
10. Rerender the image.
11. Save the image to disk.

The image saved to disk will incorporate all of the colors available in the screen mode you've chosen including the several which you picked by hand. This technique allows you to automatically mix chosen colors with ones you have manually chosen.

Merging Palettes

Another variation on the technique outlined above is the ability to merge the palettes of several pictures into a single palette which can be used to display several pictures on the same screen at the same time.

By systematically locking and freeing the palette and restricting the number of colors which can be chosen at any given time, you can extract key color information from several images to produce a palette tuned to the needs of all of the images as a group. Then, render each image using the entire palette to get even better results.

Summary

CDTV is represents breakthrough technology not because it contains whizz-bang Amiga technology, but because it bundles this technology in a package perfect for integration into the typical consumer's lifestyle.

While graphics may be important to computers...imagery is what's important to CDTV. Applications which exploit CDTV's rich imaging capability stand a significantly better chance at mass acceptance than those which do not.

Never forget, part of CDTV is quite literally TV. Mastering the techniques and addressing issues described in this article will make the imagery in your CDTV applications more vivid and true to life and therefore, make them more readily accepted.



PAL/NTSC Issues

CDTV is an international machine, available in the United States, in Europe, and other locations worldwide. The software market for this machine is therefore international as well. To reach the entire market, a CDTV title must work properly with both the PAL and NTSC television standards. Most CDTV users will have their machine connected to a television, rather than a monitor. As a result, their screen images will be on a comparatively low quality display, one in which it is especially important to pay attention to the characteristics of the television standard if an attractive (and viewable) application display is to be produced.

The basic differences between PAL and NTSC are in frequency, number of lines per screen, and the method of color generation. NTSC screen refresh frequency is 60Hz, while PAL is 50Hz. Non-overscan NTSC screens have 200 lines, while PAL screens have 256 lines. Both television standards support INTERLACE, which halves the refresh frequency (to 30Hz for NTSC and 25Hz for PAL) while doubling the number of lines that can be displayed (400 for NTSC, 512 for PAL).

Both displays can be overscanned, giving 484 lines in interlaced NTSC and 592 lines in interlaced PAL. The colors displayed will also differ somewhat between PAL and NTSC. A color that may appear as a pale magenta on a PAL system might appear as a bright pink on an NTSC system. All of these differences will present problems that must be addressed when programming for the international market in order for a title to be successful worldwide.

The first problem is the number of screen lines. It is difficult to design one screen for both 200 and 256 lines and still make good use of the display area. You may end up designing specifically for only one of the systems and converting to the other. If you design for the PAL case of 256 lines, you will have to shrink the image down when running on an NTSC system. If the system is designed for an NTSC 200 line system, the image has to be centered on a PAL system and then filled with *something* above and below the image. The optimum solution is a separate design for each system. This gives the desirable impression that the application is designed for the local market, no matter where local is.

A PAL screen requires more memory than an NTSC screen due to the additional lines on the PAL screen—56 non-interlaced/112 interlaced. This can cause problems for applications which are coded first as NTSC and then enhanced to switch to PAL. If the NTSC version uses too much memory, there won't be enough left for the switch to PAL.

Testing on both NTSC and PAL will generally show if an application has this problem. It is best to find this out early in the development process; it is difficult to squeeze 40K or 50K out of an almost completed application, especially when the rest of the company is pushing for the product to be shipped.

The difference in the number of screen lines also changes the aspect ratio of the display. It is possible to design screen images that look good under both PAL and NTSC, and it is also possible to scale according to the current aspect ratio. It is perhaps easiest, however, to keep different versions of the images for use in PAL mode and in NTSC mode.

Under V1.3, the mode can be checked by examining the PAL bit in GfxBase DisplayModes field. Under V2.0, the display database can be checked for the default monitor in the CDTV world, but it is usually more important to determine what the unit *really* is, rather than what the current default video mode. The V1.3 method, therefore, is more appropriate than the V2.0 method.

It is also necessary to establish the basic clock rate of the CDTV unit. Finding the default mode will not necessarily tell this because V2.0 allows the user to select a different video mode than the machine is jumpered for. The basic clock rate is required in certain calculations for proper timing and audio.

Graphics.library Bug For PAL. There is a bug in the V1.3 *graphics.library* which can cause the CDTV unit to come up in NTSC mode even though it should come up in PAL mode. There is an update of the CDTV OS ROM which fixes this problem. That ROM version will be available in the A690 CDTV attachment for the A500 and in later CDTV units. V2.0 does not have this problem.

If you do decide to correct the situation yourself, make sure you perform the appropriate version check. Do not attempt to change values in GfxBase under V2.0; not only is it not necessary, it is unwise as well.

The following routine can be used to determine if the V1.3 *graphics.library* made the wrong decision about PAL or NTSC. If the incorrect decision is detected, the system can be rebooted and given another chance. (Or the values in GfxBase can be corrected).

```

/*
 * PALCheck
 * Routine to test for presence of PAL. If this doesn't match
 * the system settings, try again
 */

extern struct Custom custom;

int palcheck()
{
    UBYTE agnus_chip_id;
    register int i;
    ULONG mode=NTSC; /* initial assumption */

    agnus_chip_id = (custom.vposr>>8) & 0x7f;

    if ((GfxBase=OpenLib("graphics.library"))==NULL)
        return(0); /* strange ? */

    if (GfxBase->LibNode.lib_Version > 35)
        return(0); /* got it right for 2.0 */
                    /* so we don't have to do this */

    Disable();
    if (agnus_chip_id & 0x20)
    {
        /* new hires agnus */
        /* an ntsc hires agnus has pin 41 grounded */
        /* a pal hires agnus has pin 41 open */

        if (!(agnus_chip_id & 0x10))
            mode=PAL;
    }
    else
    {
        /* an ntsc display has 262 lines, counts to 261 */
        /* a pal display has 312 lines, counts to 311 */
        /* must be run while disabled */

        if ((i = vbeamos()) > 270)
        {

```

```

        mode=PAL;
    }
    else
    {
        /* wait till vbeampos >= 256 */
        while ( (i = vbeampos()) < 256);
        do
        {
            if (i > 270)
            {
                mode=PAL;
                break;
            }
            i = vbeampos();
        } while (i > 50); /* if it falls back then no pal */
                        /* 50 is used figuring the genlock won't */
                        /* reset higher than that every frame time */
    }
    Enable();

    if ((mode==PAL) && ((GfxBase->DisplayFlags&mode)==0))
    {
        printf("ERROR: graphics opened in wrong mode\n");
        Reset();
    }
    else
    {
        printf("PAL/NTSC decision correct\n");
    }
    CloseLib(GfxBase);
}

```

The CDTV *playerprefs.library* routine **CenterCDTVView()** will provide centering information according to the user's Preferences settings, automatically taking the line difference between PAL and NTSC into account.

The 60Hz vs. 50Hz difference will generally show the most effect on timing if VBLANK interrupts are used or if the speed is synchronized to television frame update rates. If this technique is adopted, your application will run faster under NTSC than PAL. This will be especially noticeable if you time music and sound effects. Use the *timer.device* for basic timing because it compensates for PAL and NTSC. If you do your own timing, you will need to perform the same compensation if you want the application to act the same under PAL and NTSC.

The basic clock frequency differs for the NTSC and the PAL CDTV units. While the difference is not great, and for most purposes can be ignored, it does make a difference for Amiga audio, and the divisor used for directly accessing the CIA timers. Unfortunately, there is no place in the OS where this number is stored. Instead, the application must read the PAL or NTSC bit and infer the rest.

If the application is running on an NTSC machine, the NTSC clock rate divider constants should be used for audio and timer rates. If the application is running on a PAL machine, the PAL clock rate divider constants must be used. If the proper constants are not used, the pitch of the sound produced by using a specific waveform will differ between PAL and NTSC. The formulas for NTSC and PAL are as follows:

NTSC

The clock constant is 3,579,545 ticks per second.

PAL

The clock constant is 3,546,895 ticks per second.

The formula to select the period value is as follows:

$$\text{period value} = \frac{\text{desired interval}}{\text{clock interval}} = \frac{\text{clock constant}}{\text{samples per second}}$$

An example of an integer calculation using these values follows:

```
#define NTSC_CLOCK (3579545)
#define PAL_CLOCK (3546895)

ULONG clock, clockx100, periodx10, period, hertzx10, sampleBytes;

/* Check GfxBase->DisplayFlags and set clock to appropriate value
 * Set hertzx10 to 10 * desired frequency in hertz
 * Set sampleBytes to number of bytes in the one cycle waveform sample
 */

clock = (GfxBase->DisplayFlags & PAL) ? PAL_CLOCK : NTSC_CLOCK;

hertzx10 = 440 * 10; /* example frequency */
sampleBytes = 32; /* example one cycle sample length */

clockx100 = 100 * clock;
periodx10 = (clockx100 / hertzx10) / sampleBytes;

/* round off */
period = (periodx10 + 5) / 10;
```

(Also see the 2.0 *Amiga Hardware Reference Manual* for additional details in the audio hardware section. There is also a table giving precalculated period values for a five octave even-tempered scale)

Unless the proper divider constants for the system are used, the Amiga sounds will not sound quite right, especially when mixing Amiga sounds and music, and CD sounds and music (which may not be a good idea anyway). This is due to the audio DMA clock rate difference caused by the same clock frequency difference between PAL and NTSC units. However, the CD frame rate is completely unaffected by the difference in basic clock frequencies because it is the same under PAL and NTSC. This makes the CD Frame rate a convenient clock to use for timing purposes.

As mentioned earlier in the article, PAL colors do not look the same as NTSC colors. Saturated colors and pale colors are especially affected. Colors should be subdued to minimize this. Color bleeding will also be minimized through use of subdued colors.

On a saturation scale of 0 to 15, color intensity should never be greater than 13. At the same time, pale colors (such as pale pink, pale green, etc.) should be avoided as they may appear fine in one mode, but very intense or completely washed out in the other. Background colors should be off white or grey.

Testing on both PAL and NTSC television sets or monitors is really the only way to be sure an application has avoided problems. Sometimes an entirely different color scheme is needed under PAL and NTSC. While holding to the above guidelines will generally give pleasing results under both standards, testing is the only way to be absolutely sure.

Testing

A CDTV application should be tested under both PAL and NTSC systems. Attention should be paid to the appearance of the screens under both PAL and NTSC. Screens should fill the available area, and be properly centered. The feel of the application should be the same under both PAL and

NTSC. Special care will be required in selection of colors. Colors that appear subdued under one standard may appear garish when viewed on the other.

A multi-standard monitor is very useful in application testing, especially with the new CDTV units which can switch between PAL and NTSC by pushing a button. There is no substitute for trying the application under both PAL and NTSC.

Testing under PAL and NTSC should begin as early as possible during the development process, rather than left as a last minute checklist item to be performed at the end of the QA process. Some of the required changes may be involved, most notably screen design and freeing up enough RAM for larger screens. It is better not to rush these changes at the last minute. The development process will end much more smoothly if PAL and NTSC allowances are instituted throughout the development process. The end of a project is usually hectic enough as it is without finding out that without major modifications to the application, its market has suddenly become limited to only one part of the world.



CDTV Sound

The Compact Disc was designed for sound. All other elements are latecomers. CDTV discs can use CD-DA sound to its fullest extent, and if a CDTV title needs, and can afford, a symphony orchestra recorded at the highest quality, it can have it.

The Amiga computer was designed with stereo sound capabilities, and can deliver acceptable 8-bit audio on four channels, two left and two right.

A very normal decision that has to be made in designing a CDTV title is whether or not to use CD-DA sound or Amiga 8-bit sound, or both. The purpose of this article is to outline the practical considerations involved.

Here are some examples of the use of sound:

Speech

- Spoken help for users in their own language
- "Presenter" voice-overs giving information
- Dialogue
- Language teaching examples
- Narrative

Music

- Extracts for music teaching, information, or quizzes
- "Background music" for mood
- Karaoke
- Backing tracks for buskers to play along to
- Instrument samples for music creation titles
- "Fantasia"-style music with pictures and animations
- Games themes

Sound FX

- User feedback "beeps"
- Games
- Atmosphere (jungle, beach etc.)
- Spot effects for animations
- Jingles for correct or incorrect answers

The imaginative use of sound can enhance the value of a title, and also enhance its user friendliness. It can also ease the problems of creating multilingual titles. If no text is used in the title and all information is given by voice-over, the same file layout and naming conventions can be used with different presenters recorded in different languages. Switching languages, then, is simply a matter of switching directories.

Audio Feedback

There is now a wide variety of interfaces to the CDTV using infrared or wired connections. Given that the response may be uncertain, as for example, when a user covers the infrared panel on the hand-controller and presses a button, it is wise to give immediate feedback that a command has been received and understood. This is also true where an action may result in a delay; searching a large database might take many seconds or even minutes.

The simplest form of audio feedback is a keyclick. The simplest way to get keyclicks is to include *bookit.c* in your startup-sequence. Every time a key is pressed, CDTV will beep.

There will be circumstances where a beep is inappropriate: a presenter is describing something and the user wants to abort the operation by pressing button B. Here the program should decide on the appropriate response—in this instance it will probably fade the presenter out and await further instructions. Any program which is using sound to this extent will have the audio channels under its control, and will either have a sampled sound in memory ready to be played by the audio hardware, or will have set up a tiny waveform. In both cases, the sample should be resident in Chip memory at all times; reloading feedback data from the CD is not sensible.

Deciding when to beep or not can become complex, and the program code may become littered with `Beep()` commands. A far better scheme is to insert a “wedge” in the input handler and provide a conditional `BOOLEAN` for a further interrupt, which decides whether to beep or not. Precise details of this depend on whether you are using the *audio.device* or driving the audio hardware directly. However, many programmers will prefer to handle this in their input handling loop, which will take the following general scheme:

```
(* Modula-2 used as pseudo-code example *)

LOOP
  IF ButtonPressed() & ValidButton(Button) THEN
    IF BeepValid then Beep(GoodBeep) END;

    CASE Button OF

      | 0:BEGIN
        (* show a menu or picture *)
        BeepValid:=TRUE;
        Action0;
        END;

      | 1:BEGIN
        (* spool music *)
        BeepValid:=FALSE;
        Action1;
        END;

      etc.
    END; (* CASE *)
  ELSE
    IF BeepValid THEN Beep(BadBeep);END;
  END; (* IF ButtonPressed *)

  CarryOnDoingSomething;
END; (* LOOP *)
```

Ignoring the details of what constitutes the code of the two functions `ButtonPressed()` and `ValidButton()`, this scheme allows called procedures to decide whether or not a beep should be issued. An extension is the idea of *Good* feedback and *Bad* feedback. Conventional wisdom holds that Good beeps are high in pitch and Bad beeps are reasonably low. Given the sampled replay abilities of the Amiga/CDTV audio hardware they could just as well be a sigh of passion and a grunt.

When there is no important sound activity going on, audio feedback is the simplest and most efficient feedback.

Amiga Sound Versus CD-DA

The factors governing the decision as to when to use Amiga-generated 8-bit and when to use 16-bit CD-DA are:

- Quality.
- Duration.
- Access to data on the CD.

The difference between 8-bit and 16-bit sound is best explained by analogy:

16-bit sound is equivalent to a picture with 65535 horizontal lines

8-bit sound is equivalent to a picture with 256 horizontal lines

or

16-bit sound is equivalent to a picture with 65535 colors

8-bit sound is equivalent to a picture with 256 colors

For the very highest possible sound quality, it is clearly impossible for an 8-bit sound system to compete with one that has a resolution 256 times greater. The effective signal/noise ratio goes up dramatically, and the fine detail of the waveforms is far greater. A title that explores the late Beethoven string quartets is likely to require the full dynamic range of CD-DA.

In particular, quiet, sparse sound is much more difficult to capture and replay successfully on an 8-bit system because the lower signal/noise ratio makes itself apparent in hiss, and the lower granularity of the sample size tends to cause quantization artifacts, most evident where the waveform is hovering around the zero line. In this circumstance, the digitizer will have difficulty deciding whether a bit should or should not be set, thus introducing a random crackle at the very end of a dying note or spoken phrase. Noise gating can reduce this for speech, but can be unpleasant on a delicate musical phrase.

It may seem obvious to use CD-DA throughout a title, but this will reduce the total running time of a title to about seventy minutes, not allowing for graphics and other data. By comparison, a mono 8-bit sound equivalent could easily yield over eight hours of sound.

The main problem apart from duration that is associated with CD-DA is the question of access to data. While the CD-DA track is running, no further data can be drawn from the CD. This means that all pictures and data must be fetched into memory before the CD-DA is started. Nevertheless, because CD-DA play has effectively no memory requirements, almost the entire memory of the CDTV unit is available for picture and data storage, much of which can be in compressed form. It's worth bearing in mind though that to load 800K of data into memory will take at least five or six seconds, so a substantial user wait-time is generated.

For presenter speech and much music, the use of the 8-bit sound system has the advantage that the transfer rate will be much lower, allowing sound and pictures to be loaded and played in a continual stream. There is an initial load time of effectively nil, and the CD is being used as a form of virtual memory.

Acceptable sample rates for speech will lie in the range between 16,000 samples/second and 22,000 samples/second. Taking the lower figure, we can see that given a reasonable average transfer rate from the CD of 140,000 bytes/second, it is possible with good data organization on the CD to spool continuous mono sound and load and display at least one 640*200*4 picture per second.

Even mixed-mode CDs employing CD-DA for the high-quality sound examples will benefit from using Amiga 8-bit sound for all descriptive materials.

One less obvious benefit of CD-DA is ease of generation. The sound can be taken to the mastering house on DAT or reel-to-reel tape and transferred with no further processing by developers, whereas the 8-bit sound will almost certainly require digitizing by developers, and will require custom program handlers to be coded and tested. If the CD-DA sound is arranged in conventional tracks on the CD, one simple call to the CDTV_PLAYTRACK command in *cdtv.device* will produce the sound in all its richness, and if program actions need to be synchronized with the sound, the installation of a function to increment an internal software clock is easily added with the CDTV_FRAMECALL command. However, when CD-DA tracks are placed on the CD, it is advisable to leave sufficient room in track 1 (the data track) for further additions and refinements to program code and data, if you require the tracks to be very stable in their start and finish times.

A disadvantage of CD-DA is that a proof disc will be required at an early stage of development with all sound finalized, whereas 8-bit Amiga sound can be altered and tested up to the last minute.

It is clear that the decision about the proportions of CD-DA to Amiga sound must be taken at an early stage in the design process because it will govern program coding in almost all areas. It is not uncommon for publishers to demand CD-DA quality at the outset for aesthetic reasons, and then later require long animations and slide-shows to run concurrently with the sound. These two requirements may well be incompatible.

Audio Capture Tools

The Amiga is well-supplied with hardware and software to capture 8-bit audio data in mono or stereo. Almost all the currently available systems require audio data to be captured into memory. Even at modest sampling rates, the memory requirement can be substantial—two minutes of mono sound sampled at 16,000 samples/second requires nearly 2 MBytes of contiguous RAM.

When choosing a sampler, the primary consideration is quality. The Amiga has many DMA channels and interrupts running continuously, and this causes electronic noise which is easily detected by audio sampling hardware. Monitors can also inject buzz and hum into the system. Cheap samplers rarely have efficient noise filtering stages because the components are expensive and the complexity of the design goes up steeply for every decibel of noise rejection. This may not be apparent with dense, compressed sound such as rock music—it will become very obvious with speech, or a solo oboe. Samplers should be tested with low levels and quiet samples and listened to with good quality headphones before any assessment of quality can be made. Relying on demonstration samples included with the package is dangerous. Samplers that really attempt the highest quality and are powered from the Amiga will switch off interrupts and DMA before sampling; this can be detected by a completely blank grey screen during sampling.

The next consideration is the sampling length allowed. Some samplers will only permit sampling into Chip memory, and will be useless for recording blocks of presenter speech which may be far too long to fit inside what may be only 700,000 bytes of free RAM.

The most versatile samplers of all will record directly to hard disk and provide hours of continuous sound. They are likely to cost between five and ten times as much as memory-only samplers, and most of them are 16-bit samplers for the Macintosh® or PC computers.

Some samplers are mono only. This may not be a limitation for programmers, who might take a left and then right sample from tape, with a marker blip at the start on both, chop the start of both samples exactly to the marker, and merge them into a single stereo IFF 8SVX file by skipping the same number of bytes from the beginning of each of the merged files to jump over the blips. This is a useful way of doubling the effective sampling time for long stereo samples in any case.

Public domain software exists to convert Macintosh® AIFF sound files to Amiga IFF 8SVX samples. The only significant difference is the header information in the file, and programmers may find it convenient to make their sound replay routines read AIFF data directly.

A further option is to record sound in a studio equipped with 16-bit direct-to-disc equipment, transfer the files to the Amiga environment, and then software convert the data to 8-bit format. Normally, the 16-bit data will be (for Macintosh® systems) a flat binary image of the sound, in the following format:

```
WORD LeftSample
WORD RightSample
```

A very simple conversion to mono 8-bit data without oversampling will take the form:

```

MOVEA.L SampleAddress,A0 ; assume sample in RAM
MOVEA.L OutputBuffer,A1  ; where to put 8-bit data
MOVE.L SampleLength,D1   ; from file length
LOOP: MOVE.W (A0),D0       ; get left sample
      LSR.W #8,D0          ; shift MSB->LSB
      EOR.B #$80,D0        ; make signed 8-bit sample
      MOVE.B D0,(A1)+      ; store output sample
      ADDQ.L #4,A0         ; move input pointer
      SUBQ.L #4,D1         ; decrement counter
      BNE LOOP            ; until no more data
```

Normally this conversion will take place inside buffers loaded and saved from disc. The technique will perform a crude translation, and a more sophisticated version will average several samples, and also consider whether the LSB of the original sample is close to a value of 255, in which case the MSB should be incremented before the shift. You will notice that compared to the original, the converted samples have heavy bass emphasis and reduced clarity in the higher frequency range. This is to be expected, as we have just thrown away a great deal of information describing the tiny variations of the waveform. The linearity of samplers optimized for 16-bits is very different from the requirements of 8-bit sampling. To overcome this within practical bounds, record the original sound in the studio with substantial top emphasis.

With both 8-bit sampling and 16-bit sampling, the benefits of using stereo are great, arguably even greater for 8-bit sound. The difference between the left and right channels provides extra information that can help the brain to reject noise. The zero-line quantization effects become less significant and the perceived *reality* of the sound is greatly improved. Although it is more efficient in terms of space on the disc to use mono for presenter speech, for example, if it is to be heard in the context of CD-DA stereo sound, the quality will be *muffled*.

The danger zone for 8-bit sound is the threshold between low-amplitude signals and silence. Silence may be difficult or impossible to achieve. Even under studio conditions there is always some residual noise—breathing, heartbeats, air-conditioning, equipment hum—that will cause a sampler to waver.

Whenever possible, silence should be avoided for 8-bit sound. If it is feasible for a speaker to be situated in an appropriate sound landscape such as traffic, near a burbling river, in a cafe, etc., the low-level background sound will enhance the value of the commentary and mask the limitations of the audio capture.

Direct sampling of digital synthesizers may not always result in perfect results. Some synthesizers create waveforms that have inaudible partials that become clearly audible with 8-bit samplers. Only experimentation can determine what sounds good and what does not.

Good 8-bit samplers allow some adjustment to the DC Offset level. This is the internal voltage level that equates to zero, meaning the effective silence level of the sampler. If the DC Offset is not perfectly set, the sampler will produce (say) -3 instead of zero for silence. Extreme cases will cause sample clipping at one or other end of the scale, and will render editing operations to reduce noise meaningless.

One further consequence of the reduced bandwidth of 8-bit samplers is the need to keep the average peak amplitude as high as possible. An 8-bit sampler digitizing a sample at half the maximum volume becomes effectively a 4-bit sampler. It is very important to set the input level such that the full range of sound is digitized, even if at a later stage the playback routines play it quietly. It is a common mistake to think that a maximum playback volume rate (64) on the Amiga is a standard setting, and quiet sounds should be recorded quietly. All that happens in this case is that the audio hardware amplifies noise rather than signal. It is far better to record ALL sound to full amplitude and apply some discretion to the levels when playing the sound; reducing the input level degrades the sound quality—reducing the playback volume does not.

The Amiga and CDTV incorporate hardware low-pass filters to reduce aliasing of sound samples. Aliasing occurs when the sample rate is less than twice the frequency of the highest sound, and results in spurious harmonics being generated that at best, sound “interesting” and worse, result in two “beat” sounds being generated. The hardware filter comes in steeply at about 5kHz, causing a (deliberate) loss of the upper partials which masks any aliasing distortion. With high-quality samples recorded at higher speeds, the brightness of the sound may be improved by switching out the low-pass filter. The filter is a toggle, and does not have a mechanism for setting the frequency at which it operates.

The filter defaults to ON at system startup. It is controlled by a single bit (#1) in the register PRA (CIA-A Peripheral Port Data Register for Data Port A), which is at address \$BFE001. This is a read-write register, and the previous contents should be preserved when writing to it. Bit #1 also controls the brightness of the front-panel LED power display, which provides a useful visual check for the state of the filters. If bit one is set, the LED is dimmed and the filter is switched off; and vice versa:

```
PRA equ $BFE001

FilterOff: LEA PRA,A0
           BSET #1,(A0)
           RTS
FilterOn:  LEA PRA,A0
           BCLR #1,(A0)
           RTS
```

When designing sound control structures it may be helpful to include a flag to set or clear the audio filtering for this sample. When testing sounds for suitability, use high quality headphones; if no significant aliasing is detectable on these the sample should be good for most other circumstances.

Sound Editing

The basic operations sound-editing software should provide are *cut* and *save*. Most sound editing programs will do much more. They will allow you to change frequency, ramp (apply volume changes), possibly add echo, and merge and filter sounds in various ways.

In the world of 16-bit sound sampling and editing, most of the functionality of editors is dedicated to chopping and pasting. Anything else is considered best done at the recording stage.

No sound editor has yet been written that makes a badly-conceived and badly-recorded sample sound much better than it originally did. An exception to this is software to remove clicks and hiss from old recordings. Many Amiga sampling software programs provide facilities such as real-time-echo and distortion feedback as a way of *emulating* facilities which would, in the professional recording domain, be properties of the recording studio's repertoire of effects, not properties of the tape-recorder, DAT, or direct-to-disc recording system. These effects are, of course, very useful for processing instrument samples or brief sound bites.

Most sound editing software will allow you to set segments of the sample to silence. This may not be quite true, however, because what the software will do is set a segment of samples to zero. If the DC Offset or bias of the original sample was *not* zero but (say) -3, there is an increased chance of pops and clicks at the entry and exit points of the zeroed segment.

For multimedia title development, it is very likely that you will be dealing with long samples, and the editing software must be capable of handling samples longer than Chip memory.

Some operations, such as software resampling (where an existing sample in memory is adjusted to a different frequency) are very processor intensive, and if a lot of material has to be processed, a machine with a fast CPU, such as a 25 MHz 68030, will significantly speed up operations.

Most sound editing programs will have the capacity to create multi-octave IFF 8SVX files for use in music programs as musical instruments. There may be requirements for a title where heavy loading of pictures or animations from disc prohibits the use of spooled 8-bit sound or CD-DA. The ability to load some instruments from the disc and then play an IFF SMUS (or other format) music track on an interrupt system will provide an effective synchronized music track for such situations.

You Must Test On A CDTV Unit. CDTV uses a 75 frames/second interrupt system for handling CD data. This is a significant change from a naked Amiga platform. Overloading the VBeam interrupts with heavy code can easily cause the CDTV to become "interrupt bound" and the run-time system should be checked on a CDTV unit before making assumptions that it will work on a 68000 machine running with Chip memory only. The same considerations apply to reading incoming MIDI data—a 4 bitplane 640 x 200 screen uses enough system bandwidth to corrupt serial data at MIDI speeds.

Finally, when buying a sampler and sampler software, ensure that they are compatible. Most parallel-port samplers use similar protocols, but not all. For example, the high quality *Audio Engineer Plus™* from RamScan Software Pty, Ltd, is not triggered correctly by several commercially available editors.



8SVX: Playing Samples Larger Than 128K

The Amiga's audio hardware contains four digital-to-analog converters (DACs) capable of playing back digital sound samples with 8-bit resolution. On the Amiga, sound samples are usually stored in the 8SVX format which is the IFF file standard for 8-bit samples. The 8SVX standard allows samples up to 2 Gbytes long. However, the length registers of the DACs only allow samples up to 128K bytes. In order to play back a sample larger than 128K, you break it up into smaller pieces and send multiple requests to the audio device. The program listed below shows how to use this technique.

The register map in Appendix B of the *Amiga Hardware Reference Manual* shows that all the audio length registers are sixteen bits which gives a maximum length of 65,536. However, since the audio hardware can DMA two bytes at a time, the length register is set up to represent the word count of the sample, not the byte count. Hence, the maximum length for a sample is 2 times 65,536 or 128K bytes.

Audio sample data in the 8SVX format is stored as a standard IFF Chunk of type BODY. The Chunk size is given by the field `ckSize` which is a LONG variable. So the maximum sample size in an IFF Chunk is just over 2 Gbytes. Even larger samples might be possible by using a CAT or LIST. See the IFF manual for more details.

The original Amiga, the A1000, had 512K of Chip RAM, much of which would be busy doing graphics work, so the 128K audio sample size limit was a good design decision. Due to the memory limits, most early Amiga samples were short sound effects less than 128K.

More recent Amigas have 1 Mbyte or 2 Mbytes of Chip RAM and most likely additional expansion RAM. This trend will probably continue as memory prices fall making the idea of supporting 2 Gbyte samples under IFF is not so farfetched.

The program listed below shows you how to play back 8SVX samples larger than 128K on the Amiga by breaking up the sample into smaller pieces. The program divides large samples into sections of 51,200 bytes. The samples are sent to the *audio.device* using double-buffering, i.e., there are always two I/O requests in the queue. A `Wait()`, `GetMsg()` loop puts the program to sleep until the current audio request finishes.

When the program wakes up, the next request in the queue begins to play immediately. While it plays, new sample data is copied into Chip memory. The request that just ended is then reused by placing it back in the queue and the program goes through the `Wait()`, `GetMsg()` loop again. Note that there is nothing special about using 51,200-byte samples. Any size up to 128K would work but 51,200 is used to conserve Chip memory.

The IFF parsing used by the program is very simple. The FORM is first inspected for the file size and the file is read into memory. A switch statement is used to cull the VHDR and BODY Chunks from the file and pointers are set up. Other Chunks are skipped. CATs, LISTs and nested FORMs

are not supported. If the BODY Chunk has both a one-shot and continuous part, only the one-shot part is used. Similarly, only the first octave part is played in a multi-octave sample.

The *audio.device* is handled in the usual way. Two AudioIO structures and a reply port are set up. Only one audio channel is used. The channel is allocated automatically when the device is opened. The allocation key is set to accept any of the four channels. The request priority is set to 128 so that once we have the channel, it cannot be stolen by another task during playback.

An important part of the program is the setting of the clock constant to the proper value for PAL and NTSC systems. The clock constant, which is used in calculating the period of an audio request, is different on PAL and NTSC systems. To get the right value look at the DisplayFlags field of GfxBase in the *graphics.library*.

By taking advantage of the *audio.device*'s ability to queue up multiple I/O requests, it is possible to extend the effective sample size limit of the Amiga beyond the 128K barrier. The Amiga's DMA driven audio hardware can smoothly play back samples of any arbitrary size.

This program demonstrates how to play audio samples larger than 128k. An additional example using the *iffparse.library* is contained in the "IFF Source Code" section of Appendix A of the 2.0 *Amiga ROM Kernel Reference Manual: Devices*. It is called Play8SVX.c.

```

/*
 * 8SVX example - double buffers >128K samples
 *
 * Compile with SAS C 5.10  lc -bl -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/audio.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <graphics/gfxbase.h>
#include <iff/iff.h>
#include <iff/8svx.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/graphics_protos.h>

#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); } /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); } /* really */
#endif

#define VHDR MakeID('V','H','D','R')
#define BODY MakeID('B','O','D','Y')
#define MY8S MakeID('8','S','V','X')

void kill8svx(char *);
void kill8(void);

/*-----*/
/*  G L O B A L S  */
/*-----*/
struct IOAudio
{
    *AIOptr1, /* Pointers to Audio IOBs */
    *AIOptr2,
    *Aptr;
}

struct Message
{
    *msg; /* Msg, port and device for */
}

struct MsgPort
{
    *port, /* driving audio */
    *port1,*port2;
    ULONG device;
}

```

```

        UBYTE      *sbase,*fbase;          /* For sample memory allocation */
        ULONG      fsize,sbase;           /* and freeing */
struct FileHandle *v8handle;
        UBYTE      chan1[] = { 1 }; /* Audio channel allocation arrays */
        UBYTE      chan2[] = { 2 };
        UBYTE      chan3[] = { 4 };
        UBYTE      chan4[] = { 8 };
        UBYTE      *chans[] = {chan1,chan2,chan3,chan4};

        BYTE      oldpri,c;                /* Stuff for bumping priority */
        struct Task *mt=0L;
struct GfxBase *GfxBase = NULL;

/*-----*/
/*  M A I N  */
/*-----*/
void main(int argc,char **argv)
{
/*-----*/
/*  L O C A L S  */
/*-----*/
        char      *fname;                  /* File name and data pointer */
        UBYTE      *p8data;                /* for file read. */
        ULONG      clock;                  /* Clock constant */
        ULONG      length[2];              /* Sample lengths */
        BYTE      iobuffer[8];             /* Buffer for 8SVX header */
        BYTE      *psample[2];             /* Sample pointers */
        Chunk      *p8Chunk;               /* Pointers for 8SVX parsing */
        Voice8Header *pVoice8Header;
        ULONG      y,rd8count,speed;       /* Counters, sampling speed */
        ULONG      wakebit;                /* A wakeup mask */

/*-----*/
/*  C O D E  */
/*-----*/

/*-----*/
/* Check Arguments, Initialize */
/*-----*/
        fbase=0L;
        sbase=0L;
        AIOptr1=0L;
        AIOptr2=0L;
        port=0L;
        port1=0L;
        port2=0L;
        v8handle=0L;
        device=1L;

        if (argc < 2)
        {
                kill8svx("No file name given.\n");
                exit(1L);
        }
        fname=argv[1];

/*-----*/
/* Initialize Clock Constant */
/*-----*/
        GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",0L);
        if(GfxBase==0L)
        {
                puts("Can't open graphics library\n");
                exit(1L);
        }

        if(GfxBase->DisplayFlags & PAL) clock=3546895L; /* PAL clock */
        else clock=3579545L; /* NTSC clock */

        if(GfxBase)
                CloseLibrary( (struct Library *) GfxBase);

/*-----*/
/* Open the File */
/*-----*/
        v8handle= (struct FileHandle *) Open(fname,MODE_OLDFILE);
        if(v8handle==0)

```

```

    {
        kill8svx("Can't open 8SVX file.\n");
        exit(1L);
    }

/*-----*/
/* Read the 1st 8 Bytes of the File for Size */
/*-----*/
rd8count=Read((BPTR)v8handle,iobuffer,8L);
if(rd8count==-1)
{
    kill8svx ("Read error.\n");
    exit(1L);
}
if(rd8count<8)
{
    kill8svx ("Not an IFF 8SVX file, too short\n");
    exit(1L);
}

/*-----*/
/* Evaluate Header */
/*-----*/
p8Chunk=(Chunk *)iobuffer;
if( p8Chunk->ckID != FORM )
{
    kill8svx("Not an IFF FORM.\n");
    exit(1L);
}

/*-----*/
/* Allocate Memory for File and Read it in. */
/*-----*/
fbase= (UBYTE *)AllocMem(fsize=p8Chunk->ckSize , MEMF_PUBLIC|MEMF_CLEAR);
if(fbase==0)
{
    kill8svx("No memory for read.\n");
    exit(1L);
}
p8data=fbase;

rd8count=Read((BPTR)v8handle,p8data,p8Chunk->ckSize);
if(rd8count==-1)
{
    kill8svx ("Read error.\n");
    exit(1L);
}
if(rd8count<p8Chunk->ckSize)
{
    kill8svx ("Malformed IFF, too short.\n");
    exit(1L);
}

/*-----*/
/* Evaluate IFF Type */
/*-----*/
if(MakeID( *p8data, *(p8data+1) , *(p8data+2) , *(p8data+3) ) != MY8S )
{
    kill8svx("Not an IFF 8SVX file.\n");
    exit(1L);
}

/*-----*/
/* Evaluate 8SVX Chunks */
/*-----*/

p8data=p8data+4;

while( p8data < fbase+fsize )
{
    p8Chunk=(Chunk *)p8data;
    switch(p8Chunk->ckID)
    {
        case VHDR:
            /*-----*/
            /* Get a pointer to the 8SVX header for later use */
            /*-----*/

```

```

    pVoice8Header=(Voice8Header *) (p8data+8L);
    break;
case BODY:

    /*-----*/
    /* Create pointers to 1-shot and continuous parts */
    /* for the top octave and get length. Store them. */
    /*-----*/
    psample[0] = (BYTE *) (p8data + 8L);
    psample[1] = psample[0] + pVoice8Header->oneShotHiSamples;
    length[0] = (ULONG)pVoice8Header->oneShotHiSamples;
    length[1] = (ULONG)pVoice8Header->repeatHiSamples;
    break;

default:
    break;
}

/* end switch */

p8data = p8data + 8L + p8Chunk->ckSize;

if(p8Chunk->ckSize&1L == 1)
    p8data++;
}

/* Play either the one-shot or continuous, not both */
if (length[0]==0)
    y=1;
else
    y=0;

/*-----*/
/* Allocate chip memory for samples and */
/* copy from read buffer to chip memory. */
/*-----*/
if(length[y]<=102400) ssize=length[y];
else
    ssize=102400;

sbase=(UBYTE *)AllocMem( ssize , MEMF_CHIP | MEMF_CLEAR);
if(sbase==0)
{
    kill8svx("No chip memory.\n");
    exit(1L);
}
CopyMem(psample[y],sbase,ssize);
psample[y]+=ssize;

/*-----*/
/* Calculate playback sampling rate */
/*-----*/
speed = clock / pVoice8Header->samplesPerSec;

/*-----*/
/* Bump our priority */
/*-----*/
mt=FindTask(NULL);
oldpri=SetTaskPri(mt,21);

/*-----*/
/* Allocate two audio I/O blocks */
/*-----*/
AIOptr1=(struct IOAudio *)
    AllocMem( sizeof(struct IOAudio),MEMF_PUBLIC|MEMF_CLEAR);
if(AIOptr1==0)
{
    kill8svx("No IO memory\n");
    exit(1L);
}

AIOptr2=(struct IOAudio *)
    AllocMem( sizeof(struct IOAudio),MEMF_PUBLIC|MEMF_CLEAR);
if(AIOptr2==0)
{
    kill8svx("No IO memory\n");
    exit(1L);
}

```

```

/*-----*/
/* Make two reply ports */
/*-----*/
port1=CreatePort(0,0);
if(port1==0)
{
    kill8svx("No port\n");
    exit(1L);
}
port2=CreatePort(0,0);
if(port2==0)
{
    kill8svx("No port\n");
    exit(1L);
}

c=0;
while(device!=0 && c<4)
{
    /*-----*/
    /* Set up audio I/O block for channel */
    /* allocation and Open the audio device */
    /*-----*/
    AIOptr1->ioa_Request.io_Message.mn_ReplyPort = port1;
    AIOptr1->ioa_Request.io_Message.mn_Node.ln_Pri = 127; /* No stealing! */
    AIOptr1->ioa_AllocKey = 0;
    AIOptr1->ioa_Data = chans[c];
    AIOptr1->ioa_Length = 1;

    device=OpenDevice("audio.device",0L,(struct IORequest *)AIOptr1,0L);
    c++;
}
if(device!=0)
{
    kill8svx("No channel\n");
    exit(1L);
}

/*-----*/
/* Set Up Audio IO Blocks for Sample Playing */
/*-----*/
AIOptr1->ioa_Request.io_Command = CMD_WRITE;
AIOptr1->ioa_Request.io_Flags = ADIOF_PERVOL;
/*-----*/
/* Volume */
/*-----*/
AIOptr1->ioa_Volume=60;
/*-----*/
/* Period/Cycles */
/*-----*/
AIOptr1->ioa_Period =(UWORD) speed;
AIOptr1->ioa_Cycles =1;

*AIOptr2 = *AIOptr1; /* Make sure we have the same allocation keys, */
/* same channels selected and same flags */
/* (but different ports...) */
AIOptr1->ioa_Request.io_Message.mn_ReplyPort = port1;
AIOptr2->ioa_Request.io_Message.mn_ReplyPort = port2;

/*-----*/
/* Data */
/*-----*/
AIOptr1->ioa_Data = (UBYTE *) sbase;
AIOptr2->ioa_Data = (UBYTE *) sbase + 51200;

/*-----*/
/* Run the sample */
/*-----*/
if(length[y]<=102400)
{
    AIOptr1->ioa_Length=length[y]; /* No double buffering needed */
    BeginIO((struct IORequest *)AIOptr1); /* Begin the sample, wait for */
    wakebit=0L; /* it to finish, then quit. */
    wakebit=Wait(1 << port1->mp_SigBit);
    while((msg=GetMsg(port1))==0){};
}

```

```

else
{
    length[y] -= 102400; /* It's a real long sample so */
    AIOptr1->ioa_Length = 51200L; /* double buffering is needed */
    AIOptr2->ioa_Length = 51200L;
    BeginIO((struct IORequest *)AIOptr1); /* Start up the first 2 blocks... */
    BeginIO((struct IORequest *)AIOptr2);
    Aptr = AIOptr1;
    port = port1; /* Set the switch... */

    while(length[y] > 0)
    {
        /* We Wait() for one IO to finish, */
        wakebit = Wait(1 << port->mp_SigBit); /* then reuse the IO block & queue */
        while((msg = GetMsg(port)) == 0){}; /* it up again while the 2nd IO */
        /* block plays. Switch and repeat. */

        /* Set length of next IO block */
        if(length[y] <= 51200) Aptr->ioa_Length = length[y];
        else Aptr->ioa_Length = 51200L;

        /* Copy sample fragment from read buffer to chip memory */
        CopyMem(psampl[y], Aptr->ioa_Data, Aptr->ioa_Length);

        /* Adjust size and pointer of read buffer */
        length[y] -= Aptr->ioa_Length;
        psampl[y] += 51200;

        BeginIO((struct IORequest *)Aptr);

        if(Aptr == AIOptr1)
        {
            Aptr = AIOptr2; /* This logic handles switching */
            port = port2; /* between the 2 IO blocks and */
        } /* the 2 ports we are using. */
        else {Aptr = AIOptr1;
            port = port1;
        }
    }

    /*-----*/
    /* OK we are at the end of the sample so just wait */
    /* for the last two parts of the sample to finish */
    /*-----*/
    wakebit = Wait(1 << port->mp_SigBit);
    while((msg = GetMsg(port)) == 0){};

    if(Aptr == AIOptr1)
    {
        Aptr = AIOptr2; /* This logic handles switching */
        port = port2; /* between the 2 IO blocks and */
    } /* the 2 ports we are using. */
    else {Aptr = AIOptr1;
        port = port1;
    }

    wakebit = Wait(1 << port->mp_SigBit);
    while((msg = GetMsg(port)) == 0){};
}

kill8();
exit(0L);
}

/*-----*/
/* Abort the Read */
/*-----*/
void
kill8svx(kill8svxstring)
char *kill8svxstring;
{
    puts(kill8svxstring);
    kill8();
}

/*-----*/
/* Return system resources */
/*-----*/
void
kill8()

```



```
{
if(device ==0) CloseDevice((struct IORequest *)AIOptr1);
if(port1 !=0) DeletePort(port1);
if(port2 !=0) DeletePort(port2);
if(AIOptr1!=0) FreeMem( AIOptr1,sizeof(struct IOAudio) );
if(AIOptr2!=0) FreeMem( AIOptr2,sizeof(struct IOAudio) );

if(mt!=0)      SetTaskPri(mt,oldpri);

if(sbase !=0) FreeMem (sbase, ssize);
if(fbase !=0) FreeMem(fbase, fsize);
if(v8handle!=0) Close((BPTR)v8handle);
}
```

Producing High Quality Digitized Multilingual Narrative Audio

Overview

With a storage capacity of over 660 megabytes, Commodore Dynamic Total Vision (CDTV) naturally lends itself to programs which incorporate digitized audio narratives. These digital audio cuts can be used to interact at a high level with the user in his own language as he moves through the application, offering commentary, conversation, information and help which will enhance the CDTV experience.

Since CDTV will be marketed in European and Japanese markets, the developer should consider providing audio narration in multiple languages. All of the applications planned or completed for CDTV will contain digital audio narratives in English, German, Japanese, French, Spanish and Italian.

The CDTV title, *Classic Board Games*, provided an opportunity to come to terms with the problems of producing and maintaining relatively high quality digitized audio narrative. It is the purpose of this paper to share this knowledge with other CDTV developers, so that a high level of audio quality might be maintained in all CDTV applications. High quality audio should present a more appealing image to the consumer and lead to more sales of CDTV units and software.

Digital Sampling Facts and Figures

It is suggested that the CDTV software developer be familiar with the Audio Device chapter of *Amiga ROM Kernel Reference Manual: Libraries and Devices*, and the Audio Hardware chapter in the *Amiga Hardware Reference Manual*.

Rate, Time and Memory are the three factors that need to be considered when playing a digital sample.

- *Rate* indicates the number of 8-bit samples per second that comprise a digital cut. The higher the sampling rate, the higher the quality of sound produced.
- *Time* is the length of a given digital cut in seconds.
- *Memory* is the number of bytes required to store the sample in RAM.

Producing High Quality Digitized Narrative CDTV Audio Cuts

Some magic numbers to consider:

- The Amiga (CDTV) is limited to playing 28867 eight bit samples per second from each of four audio channels, two of which will play from the right audio port and two from the left audio port, so that stereo output may be produced.
- NTSC and PAL systems possess different clock values that must be considered during playback: 3579545 for NTSC, and 3546895 for PAL. The clock value needs to be taken into account before playing back any audio samples.
- The maximum size of a sound sample is 128K (131072) bytes. Samples longer than this must be spliced together using double buffering techniques described in the "8SVX: Playing Samples Larger Than 128K" article.
- The highest frequency you can record is determined by the sampling rate divided by 2.

Scripting

The first phase in producing multilingual digital samples is scripting. The script needs to be written with the Magic Numbers mentioned above in mind. For instance in scripting *Classic Board Games*, tests showed that two 128K sound buffers should be used in the program. A sampling rate of 16273 samples per second because it would provide a maximum of eight seconds of continuous sound from each buffer, and because it allowed frequencies up to 8000 KHz., which would be adequate to reproduce the voices of all the male narrators who were to be recorded.

Had female narrators been used, the sampling rate would have had to be increased to allow for the higher frequencies produced by the average female voice.

After the script is written, it is divided into passages that can be spoken in seven seconds or less, making the breaks at punctuation points whenever possible. Also, each passage is given a slate code.

If a passage is short, then it is given a single slate code, while longer passages might require 2, 3 or 4 slate codes. It is necessary to break up the passages, because there is often no blank space in a spoken sentence which can be used to divide passages. Short passages are also easier to mix and order into different combinations within the software.

Passages of a large script have to be translated, handled and processed in multiple languages, some of which (Japanese) are written with non Greco-Roman alphabets. The slate codes are used for passage identification, and are later used as file names for each digital sample. For example, file D04E would contain the English sample, file D04G the German, file D04J the Japanese, and so on.

Script Excerpt from *Classic Board Games*

D-4 "The New Game option begins a game,"

D-4 _____

{pause}

D-5 "while the Continue option resumes the game that was just completed."

D-5 _____

{pause}

D-6 "The Replay option automatically re-plays the game that was just complete."

D-6 _____

How Long is a Pause? The {pause} is a one second hesitation on the part of the narrator during the taping session.

Translations

The translators were asked to translate the script considering the following stipulations.

- The start of each passage is designated by a boldfaced, italicized slate code (e.g., **D-4**), which is the passage ID.
- Write the translation of a passage on the lines immediately following the passage (two lines for each slate code). Each translated passage following a slate code should take no more than 7 seconds to speak.

Sound Studio: The Taping Session

The *Classic Board Games* crew was fortunate to have located a sound engineer who specializes in taping vocalizations—including multilingual vocalizations, and who has a studio at his disposal that is designed to record individual dialogue. A good sound engineer will assure your recordings are of high quality.

The digital samples played on the Amiga from the first taping session tended to have too much bass, perhaps due to the lack of overtones above the 8000 KHz level. To compensate for this, a low end 80 KHz cutoff was used on the studio board to compress the sound as it was laid on the tape. While it is also possible to implement high end compression, the sound engineer advises against it, as it tends to flatten the tones.

Signal strength should be carefully monitored to maintain a constant volume level. Scaling the sample in the computer to adjust the volume level between samples, lowers the quality of the sample. Adjustments in all aspects of audio are best made in the studio.

Volume level is important because digitized passages might later be played in any order within the application. Narrators tend to fluctuate their speaking volume from page to page when reading as their energy level varies. Narrators also tend to vocally project more at the start of a page than at the end of a page, and during series of monotonously repetitive passages, vocal projection fades (perhaps from boredom). When any of these situations occur, retakes are necessary. Otherwise when passages are played in any order other than that at which they were recorded, volume variations will be apparent.

Keeping a constant distance between the mouth and microphone of the narrator helps maintain an even volume level. Some narrators may tend to dip their head when reading to the bottom of the page, which inadvertently increases mouth to mike distance. Advise the narrator to read with his eyes, or move the paper upwards, as he reads down the page so mouth to mike distance is maintained.

The sound engineer recommends that each narrator wear a headset to monitor his own voice level as he speaks. This seems to be an excellent idea. One of the *Classic Board Games* narrators did not wish to wear the headset and there was more trouble maintaining a constant volume level with him than with any of the other narrators.

During recording it is necessary to listen carefully for any hesitation or fluctuations in the narrator's voice. Different pronunciations of the same word from one passage to another must also be guarded against, as in "you say tomahto and I say tomayto". This can be exceedingly difficult to monitor when the narration is in a foreign language.

Voice intensity, crispness, clarity, etc., should be monitored carefully. When in doubt, do a retake. Ask foreign language narrators to also monitor themselves and request a retake whenever they are not satisfied with their performance. A script reader should also be following the narrator's words, checking for any errors, variations, or inadvertently deleted words.

You may wish to record multiple takes of each passage, so that you can choose the best one later. It seems that the first take is generally better than subsequent takes, unless one of the problems mentioned above dictates a retake. Multiple takes are also going to proportionately increase the time and cost of the session. If you decide on single takes, have the engineer rewind the tape before doing a retake so that only good takes reside on the recording. This will dramatically decrease the time spent digitizing each cut.

In the *Classic Board Games* taping sessions, the sound engineer inserted an even tone between each take, which was later helpful during the digitizing process. These tones are recognizable on the sound sample graph and act as both visual and auditory markers between cuts.

An excerpt from one of the *Classic Board Games* taping sessions went as follows:

Engineer:	{tone on tape} {voice on tape}	"A1"
Narrator:	{voice on tape} {tone on tape}	reads A01 text
Engineer:	{voice on tape}	"A2, A3, and A4"
Narrator:	{voice on tape}	{reads A02 text} {pause} {reads A03 text} {pause} {reads A04 text} {tone on tape}
Engineer:	{voice on tape}	"B1"
Narrator:	{voice on tape}	reads B01 text {tone on tape}

All cuts should be timed as the recording is made. Whenever a given cut takes longer than 7 seconds, retake the cut with the narrator picking up the pace a bit. During the *Classic Board Games* recording, only German and Italian required the occasional time related retake. For what it's worth, the following table indicates the relative sizes of all narrative cuts as translated in each language in *Classic Board Games*. This information may be of use, if you are trying to pre-calculate disk space requirements for an application with a large amount of audio data.

English	14.0 MBytes	German	19.0 MBytes
Japanese	13.7 MBytes	French	15.0 MBytes
Spanish	16.0 MBytes	Italian	18.6 MBytes

Digitizing

A *FutureSound* stereo digitizer in monaural mode was used to digitize narrative cuts. Any of the digitizers on the market will probably work as long as they have a gain control. You may digitize in stereo, but keep in mind that this will require twice as much space to store the files and double the loading time.

As previously mentioned, all adjustments to the sound cuts should be done in the studio. It is recommended that no adjustments be made in equalizing tones, noise filtering, or whatever between tape player and digitizer. Further, you should avoid using Dolby because it seems to have the counterproductive effect of inducing noise into the digitized sample rather than removing it.

The best results seem to be produced by connecting the Tape Out jacks on the tape machine directly to the digitizer via a shielded cable. Anything you put between the recorded source and the digitizer may effect the sample adversely.

Use the gain control on the digitizer to adjust the input level. Setting the gain somewhat high seems to produce the best results, possibly because it minimizes the background noise level. This does produce some minor clipping, however, which will have to be repaired either manually or by means of software.

A clip is a spot in the sound sample where the value of the sound exceeds the maximum value of +127 and produces a spike value of -128, that is audible as a popping noise. Minimal clipping can be repaired by converting the -128 spike to +127 wherever it occurs. At a sampling rate of 16273, repaired clips of less than 3 consecutive bytes sound perfectly normal when played.

Whether you record a tape or DAT, background noise, or hiss, will be present, but it should be minimal. The best you'll probably be able to do is maintain a background noise level of 1 in all bytes within a blank space of a sample where no one is speaking (a 0 would be no noise). Such a low level hiss is audible only when listening to a digitized sample with treble set high and base set low. Under normal tone settings blank space bytes containing a 1 are not audible.

Digitize as many seconds of audio as you can, given the memory limitations of your system. Most digitizer software will allow FAST memory to be used when sampling, which will allow large samples to be grabbed from the recording. You may want to keep your samples in Chip memory so that playback and editing can be done within the environment in which the final product will be played. Also, using a headset when digitizing and editing narration allows you to carefully study each sample for imperfections.

When marking the starting point of a sample cut from a larger sampling, be sure not to cut off any bytes that should be included in the sample. Being able to view a graphical display of each sample is imperative, especially in zoom mode, as it will allow you to see faint values that you may otherwise miss. Particularly with foreign language cuts, listen carefully to the playback while reading the text to be sure that all initial letter sounds are present.

A good suggestion is that punctuation spacing be included at the end of each sample cut so that the playback routine within the application does not have to maintain delay values for each cut. For example, in English, a period may require 2.0 seconds of blank space, a comma .8 seconds and a semicolon 1.25 seconds. If this blank space is appended to cuts ending in punctuation marks, the playback routine can just load and play samples consecutively. Be aware, however, that punctuation blank time varies with each narrator, the rate at which the sample is read, and also the language spoken. Punctuation blank times should be modified to the speaking pace of the narrator and the language.

Finally, it goes without saying that you must be extremely careful to save each narrative cut under the correct file name in the appropriate directory. The slate system outlined in this paper was designed to minimize the error potential from this process. But with potentially hundreds of cuts in each language, the odds of errors occurring are increased, so caution is still advised.

Turning Off the Audio Filter

It is advisable to turn off the audio cutoff filter when narration is played within your program. Turning off this filter will allow higher frequency overtones (up to 15 KHz) to be heard. The effect of turning off the filter on a narration cut is to produce a clearer, less muffled tone. Dan Schein wrote an excellent Amiga Mail article ("Amiga Audio Cutoff Filter", Volume One, Page VI-23) on the subject that is recommended reading.

Conclusion

It is hoped that this paper will facilitate the production of a minimum quality standard in digitized narration in all applications written for CDTV. It is assumed that, as we all gain experience in writing CDTV applications, better methods of producing CDTV digitized narration will be developed.

Of course, the highest quality sound will always be CD Audio played directly from the disc. With only 72 minutes of CD Audio available on a disc, however, this method of producing should would limit an application supporting six languages to but 12 minutes of audio per language. With CD Audio it is also impossible to access the disc to load graphics or animators while CD Audio is playing. (Except by using special techniques. *Subcode data may be useful in some applications for overcoming this.*) It is, therefore, very likely that digitized audio will continue to be an important feature in many future CDTV applications.



CDTV Audio Cookbook

Lead In

What follows is a brief account of the main factors involved in producing a successful Mixed Mode CD. It comes from some very late nights and some very inelegant programming on the Music Maker project. It is not definitive. Every programmer will have a specific need and a specifically economical route to it. All code here is in pseudo-code, because Amiga programs are now written in variety of languages—C, assembler, Amos, Modula-2, and Oberon (the first efficient Object Oriented language available on the Amiga.) This article assumes that the reader is familiar with the use of the Amiga devices. If not, it is highly recommended to learn them before committing a product to a very expensive circle of plastic.

Audio Play

A Mixed Mode CD has up to 99 tracks on it. A track is really only a logical division of the total CD data into subsections. Although the structure of sectors in a data track is different from that in a CD-DA track, there is no need to give it even a second's thought.

On a Mixed Mode disc, track one is always data—most times this is an AmigaDOS image sitting inside the standard CD-ROM file system known as ISO 9660. Playing track one on an audio player can produce anything from an impromptu KraftWerk experience to speaker cones vanishing down the street. The other tracks can, of course, be anything from a few seconds of music to a complete Mahler symphony, but effectively, the disc space can be thought of in terms of time.

A CD typically contains 72 minutes of time. That time can be subdivided into data time and audio time. Given that data gets eaten at the rate of 150K per second, a minute's worth of time is approximately 60×150000 bytes or 9,000,000 bytes. Assuming that you have 53 minutes of audio, you can see that you are left with about 19 minutes or approximately 170 megabytes for your data track. Simply put, more audio means less data and vice versa.

There are at least two commands for playing CD-DA tracks using the *cdtv.device*. One is the **PLAYTRACK** command and the other is the **PLAYMSF** command. **PLAYTRACK** is used for most purposes because it doesn't matter if your track wanders about the disc in various iterations of the development loop. If it's track 8 it will always be track 8.

PLAYMSF is used for precise start times. The PlayTrack example in the CDTV Developers' notes gives a perfect example of using **PlayMSF**.

To play a track, whether using **PLAYTRACK** or **PLAYMSF**, you do the following steps:

1. Create a message port.
2. Create an I/O request structure.
3. Open the *cdtv.device*.

4. Fill in the I/O request structure with the appropriate values:
 - a) set the `io_Command` field to either `PLAYTRACK` or `PLAYMSF`
 - b) set the `io_Offset` field to the starting track for `PLAYTRACK` and the starting time for `PLAYMSF`
 - c) set the `io_Length` field to the stopping track for `PLAYTRACK` and the ending time for `PLAYMSF`
 - d) set the `io_Data` field to 0
5. Either `DoIO()` or `SendIO()` the I/O request structure.

Terminating Conditions

The audio track will play until it finishes if you don't stop it. If you used the synchronous `DoIO()` to send the command, you will not regain control till the track is done, so there is no way to stop a track that is playing. That's easy enough. The complexity comes in when you use the asynchronous `SendIO()` to send the command. In that case, you do have the option of stopping it before the ending track or time you specified in the I/O request.

There are two ways of stopping a track. You can either `AbortIO()` the I/O request or send the `STOPTRACK` command and then `WaitIO()` the initial request. Regardless of the method you use, you must know whether the I/O request is still active before attempting to stop it otherwise you'll create some fireworks. You do this by using `CheckIO()`.

Making certain that an I/O request is active before attempting to stop it is called protective coding. *Check* that you have closed any files *before* playing the audio (there could be a file read still pending) and *check* that you have stopped the track before doing anything involving the CD audio or file system. It's a very easy trap to get into and it has serious consequences.

If you take those steps, playing audio tracks is easy—providing that the track exists, of course.

Track Checking

Normally, you will know how many tracks you have on a disc, but suppose you have a condition where the user can pop the CDTV disc and stick an ordinary CD-DA disc into the drive. In that case, you need to know if the disc is valid for your application and if so, how many tracks it has.

You can determine this using two commands, `ISROM` and `TOC`. The `ISROM` command tells you if the disc is a CD-ROM and the `TOC` command returns the disc table of contents in two formats, logical sector number and time.

If a user inserts a CD-ROM disc when you expect a CD-DA disc and you don't use `ISROM` to determine the type of disc, you could end up playing track ONE (the data track of a CD-ROM) and chew someone's loudspeakers. It's always a good idea to use `ISROM` whenever your application uses more than one disc.

Sending the `TOC` command specifying track ZERO tells you the first and last track, from which you can limit the number of tracks the user can attempt to play. It's arguably a user's own fault if he starts popping discs during a normal program run when that's outside the rules of the game, but you have to make things idiot-proof. Remember also that the CD-DA side of the system and the

file system side are sharing the same drive, so if he *does* pop the disc you'd better check that your disc is back in before even thinking about opening a file.

A crude method for doing this: if you do a TOC call and find that ISROM returns TRUE and the disc has the right number of tracks, assume it's the right one. Remember, we did say crude.

A more sophisticated version: do the same as above, but also check for your manufacturer's ID and product ID. This is the safest course of action.

PLAYMSF

The PLAYMSF command plays from one time to another time. This is where you face the great dilemma...where are the tracks?

To keep the software update and test cycle to the minimum you must work closely with the CD manufacturers. There is a big difference between mastering what are essentially Amiga programs on a CD and mixed mode discs. If you are going to synchronize anything to an audio track, it is *vital* that the tracks not drift between cycles in the development. And they can. Go for a good manufacturer because they will create a master tape that leaves the tracks (after initial editing, see later comments) starting and stopping at almost exactly the same frame. Ask for your disc to be laid out such that the audio tracks are at the end, leaving a large data track space into which your updated data and program images will be placed. If you don't do this, you will suffer from wandering track syndrome which can be expensive.

For many applications, it is appropriate to create a CD with the audio tracks placed at the end and nothing in track ONE, which could be several minutes of silence. In other words, if you leave a big enough hole for your data you can go through many iterations of replacing the data track whilst leaving your audio in exactly the same physical space on the disc.

You can find out where your tracks are by (a) reading the TOC or (b) getting a printout from the manufacturer. These give you a crude idea. You can then experiment with your program (or, to be more sensible, a control file) and set an exact time to start each track from. You are now independent of the disc layout, lead-ins, any silence from the transfer from DAT or analog tape. Unfortunately, you are also crucially vulnerable to wandering tracks.

The PLAYMSF command uses the MSF format. As it is listed in the *cdtv.h* include file, the MSF format looks a bit obscure with all those shifts, but what it boils down to is simple: a LONG (32 bit) word with the three least significant bytes holding M (minutes), S (seconds) and F (frames), or to put it in a civilized language:

```
TYPE MSF=RECORD dummy,minutes,seconds,frames:BYTE END;
```

The dummy is there simply to force the right-justification. The *cdtv.h* information is simple and accurate about this. Set up the I/O request, SendIO() or DoIO() it, and enjoy the speed at which the track comes up.

Nothing in life is simple, so there are a few potential nasties here. As it says in the *cdtv.h* file, *just aborting* a PLAYMSF command is not enough because the disc light stays on and the laser is still active. Here is an example where you *must* use the STOPTRACK command even if the track has in fact stopped, unless you want to pretend you're doing full motion video of course, but then, who isn't (pretending)?

How Do I Know When It Has Started?

It's unwise to assume that the system will respond instantly just because a command is issued. The CD system has a lot of indeterminacy in it. For example, the laser may have been repositioned at the far point on the CD. For tight synchronization to a CD-DA track, you must therefore wait until the audio is definitely playing. This is a case for a tight loop using the CDTV_QUICKSTATUS command to *cdtv.device*. For audio purposes we are only really interested in one bit in the longword returned in the IOStdReq structure's *io_Actual* field—the "audio" bit, defined as QSB_AUDIO, or bit 2. You set up the IOStdReq with the *io_Command* field containing CDTV_QUICKSTATUS, DoIO the request, and get lots of bits back in the *io_Actual* field.

For tight synchronization, the method will look a something like this (in Modula-2), assuming that the CDTV_FRAMECALL has been used to install a little code section that increments the variable "timeCount" 75 times/sec:

```
CONST TooLong=150;
VAR x:INTEGER;
(* etc *)
    timeCount:=0;
    REPEAT
    x:=DoIO (ADR(MyQuickRequest));
    UNTIL (
        (x#0) OR
        (timeCount>TooLong) OR
        (QSB_AUDIO IN LONGBITSET(MyQuickRequest.io_Actual)
    );
```

In other words, if bit 2 (QSB_AUDIO) is set, audio is issuing forth in all its glory from the CD.

At this point we can test *x* and *timeCount* and if they are both looking healthy, we can now synchronize to the audio by setting *timeCount:=0*, and assume that the audio is running and so is our clock.

If you don't need this kind of frame-locked accuracy, this type of coding is a little heavy. If you are synchronizing an animation or a MIDI replay to some audio, it is vital to know when your command to play some audio is up and running.

When a title allows the users to pop CDs and insert any CD they wish, the time-out will be essential. When this call fails you may be facing a massively corrupt CD, no CD at all, a data track, or a non-existent track; this failure may be a good time to read the Table Of Contents on the CD and take appropriate action.

Synchronizing to the CD-DA

The track is running, now what? You have a file, say from a music sequencer, in MSF format, starting from an offset of 0:0:0 into the track (improbable, but you get the point)—how do you lock it to the CD-DA?

The best way is to use the FRAMECALL command. This tells the system to activate a handler function every time a new "frame" of audio data is sent to the audio output. CD frames occur 75 times per second. Unfortunately, like the habit of driving on the "wrong" side of the road, SMPTE timings come in various national flavors. SMPTE frame rates can be (normally) 24, 25 or 30 frames per second. It's up to you, or more accurately, up to your parents.

Getting 25 frames a second is easy—your handler code counts in threes and updates a clock in MSF format remembering that the frames will then run from 1–25, after which you increment seconds. The calculation for 24 or 30 frames is nastier and is left to you. The danger point here is that the FRAMECALL handler happens on the supervisor stack. You had better not get fancy with it! Just increment a master clock MSF variable or Cause() a software interrupt.

The gotcha factor in the FRAMECALL handler is that you SendIO() the I/O request and do nothing else with it until you want to kill it, when you AbortIO() it and *wait until its terminated*.

Having set this up, you are now in a position to check your internal data structures against the master clock which is being incremented by the framecall and do things as and when required.

If you need to use more precise timing, you can use one of the CIA timers for microsecond precision (taking PAL/NTSC into account) and maybe adjust a fudge factor by periodically checking the framecall result against expectations.

One Timer Does Not Fit All. System use of the CIA timers varies under different versions of the operating system and even changes dynamically under 2.0 (the 2.0 *timer.device* attempts to accommodate any application that properly tries to allocate any CIA timer). You should always attempt to allocate *one* of the CIA timers instead of hardcoding in a *particular* timer. See the *cia.resource* section of the “Resources” chapter in the 2.0 *Amiga ROM Kernel Reference Manual: Devices* for an excellent example of allocating and using an available CIA timer.

Generally, it's easier to pre-convert standard SMPTE timings into 75 frame/second CD frame rates. In fact, as a general rule on CDTV, it's better to pre-compute everything you can because you have loads of data space to store the computations as opposed to having a slow processor running in Chip RAM do the computations at run-time. (For this reason, ACBM format or a variant is a much faster way of loading and showing pictures than using compressed ILBMs or, at least for HAM or EHB photo pictures, the debbox routines.) Unpacking a standard MIDI file format in real time on the CDTV is not sensible. It is much better to write a conversion program and massage the data into a quick-load format. A high proportion of your development cycle will be involved in writing tools to maximize data efficiency.

For example, suppose you have a MIDI file which has one track running at 60 ppq with 192 MIDI clocks per crotchet. To convert the “events” in this file to CD-frame timed format you can see that one crotchet lasts exactly one second, and if the first starts at 0 frames, the second will start at 75 frames and the third at 150 frames. You will conclude from this that the absolute frame time for any event is a simple conversion of the total elapsed time from the start of the playback. This is fine for tracks without tempo variations, but when there is a tempo change you have to do a bit of juggling with your maths, i.e., reset your logical master clock to zero, compute any remaining time on any track event back to MIDI clocks, and then redo the sums.

The great advantage of doing this offline prior to mastering is that you can test it to destruction before committing to the expensive plastic.

PAL—NTSC

The PAL-NTSC conflict remains one of the biggest problems on the Amiga/CDTV. Many of the American developers don't really understand PAL because they don't have to worry about it and make dangerous assumptions. One of the great advantages of the CDTV is that the CD frame rate is the same under both PAL and NTSC, whereas everything else, VBeams, DMA, processor clock, etc., is different. So unless you *really can't*, use the CD frame rate as a clock and not, well, as a clock!

An NTSC machine is an NTSC machine. Period. A PAL machine may end up (about 50 percent of the time with a genlock board in!) thinking it's an NTSC machine when it isn't. The gfxBase flags are *totally and utterly unreliable* about this. The best way to determine the type of machine is by the method described in the "PAL/NTSC Issues" article.

The audio DMA clock rate is also different between the two systems. Usually the difference is barely audible if you don't have perfect pitch, but if you have CD-DA running a music track and do not check that you are really on a specific model and use the correct the clock divide constant, the track will sound bad.

The clock constants for period calculations are 3,579,545 clocks/second on an NTSC machine and 3,546,895 clocks/second on a PAL machine.

Other Audio Considerations

First, a few common questions:

(Q) Can I play just the left or right CD-DA, hence doubling the mono high-quality playback time to 144 minutes?

(A) No.

(Q) Can I get at the CD-DA 16 bit DACs and play back my own 16 bit internal samples under program control?

(A) No.

(Q) Can I get the CD-DA data into memory?

(A) No. The CD-DA side of the system is sealed for licensing reasons.

(Q) Can I alter the CD-DA volume?

(A) Yes. The MUTE and FADE commands fully control CD-DA volume level.

(Q) Is the CD-DA audio output level locked to the internal sound generation circuitry?

(A) No. Forget theory about relative dB levels and set your volume ratios when you have a test disc. Theory is expensive in CDTV development.

Audio Quality

There are two (or more) issues about audio quality. The first is for CD-DA tracks, the second for internal 8-bit sounds.

Using CD-DA is pointless if you do not produce a professional quality recording, from which it follows that waving a cassette recorder in the air is going to give you some of the most expensive hiss you can reproduce. DAT recorders work to the same specification as CD-DA and what you hear is what you get.

However, do not make the assumption that either DAT recorders or CD units work at an invariable speed. Speed variations between different drives can be significant—like 2 minutes over 72 minutes, meaning a potential closing velocity of four minutes between two drives! You may run into problems if you rely on stopwatch timings for long files. So the way to cut down problems is: lay the audio tracks, get them edited by the CD manufacturer onto the master tape, make a test pressing and leave them alone.

It is obviously outside the scope of this document to describe audio recording techniques. Suffice it to say that for professional results there is no substitute for a professional recording studio and engineer.

But why use CD-DA? For many purposes it is not necessary. It is expensive, and for simple speech files, etc., may chew up your disc at an alarming rate. If you want professional quality music there is no alternative. For help files, commentary, the ordinary traffic of multimedia, you may as well use Amiga audio. Bear in mind, though, that you will frequently have to use a background spooling process to play long samples (remember the 128K limit for DMA samples, although there is a workaround) and this may eat some processor time. The CD-DA is fully asynchronous and leaves you to do whatever you like.

There is a wealth of example programs in the public domain which show you how to play Amiga audio, so there is no need to go deeply into that. What is rarely mentioned, though, is the invidious comparison between 16-bit and 8-bit sound that cannot be avoided when you have a disc with CD-DA and 8-bit sounds running consecutively or together. Most Amiga sound demos use rock music and sound pretty good; most CDTV applications use speech which may not.

Rock music is usually densely-textured and compressed. Audio compression means that the overall signal strength is forced up to something near a constant level. This is very appropriate for the audio hardware because the noise is effectively buried in the signal. With speech, however, there are very low level signals at the ends of words. Here the amplitude of the waveform drops off until it is hovering around the zero line. In other words, it is revealing the inescapable truth that 8 bits gives you vastly less precision than 16 bits. The low level sounds may crackle or break up around the zero amplitude line as the sampler tries to decide whether this tiny noise level constitutes 1 or 0. Most good sampling software will allow you to ramp the amplitude down a bit when you have made the sample. This may well knock out some rogue bits, but will also introduce new errors at the next level down. You can keep on doing this until you have silence.

The simplest solution is to use a gate in the recording studio. This electronic filter will trap any sound in the danger zone around the zero level and not pass it out of the mixing desk. Gates should be used with caution though, as they can cause a nasty clipping if the threshold is too high. Just a little gate should be enough. Using a microphone with very high sensitivity to the upper range of frequencies is more likely to accentuate exactly the areas you want to eliminate. What we are

doing here is trying to optimize the recording for the hardware on which it will run, not strive for the perfect recording, which is the CD-DA path.

You may have problems with some recording engineers—well, most people have problems with recording engineers. The engineer is trained to get the highest possible bandwidth from his studio, but you will want him to cripple his system so that the final result will sound better on the CDTV than his perfect recording would. If necessary, equalize the top frequencies down a little, avoid sibilance like the plague, and watch out for the wavering zero-line quantization problem.

For a first session or tests, it is a good idea to go straight from the desk into an Amiga digitizer. Use the best you can get. Some of the cheaper units work fine at high amplitudes but have poor internal noise rejection—they will be slinging loads of Amiga bus noise into the ADCs.

When working with audio, it is important to remember the wide range of equipment performance that will be used by CDTV owners when they play the disc. You could be on a dreadful television or a megabuck hi-fi system. The acoustics of your room will color the sound you hear to a very significant degree, so the best method for good quality control is to take the audio output either from the CDTV headphone jack, or on an Amiga via an amp or mixer and use a pair of high quality headphones. They will defeat the room's acoustics and give you good sensitivity to pops, crackles and noise. They may be too harsh, in fact, but that's better than simply not hearing the problem.

There are other methods of getting digital sound into the system. On Music Maker, the instruments were sampled in the studio on the 16-bit Casio S1000 sampler, shaped (on an ST) with Avalon, saved to disc, passed to an Amiga, and then computed from 16 to 8 bits with a custom program tool.

There are DAT→disc systems available on some platforms (Mac, PC, ST) and any time now on the Amiga. Their success or failure in this realm of application-building depends entirely on how good they are at converting from 16 to 8 bits. They do not eliminate the zero line quantization problem unless they are very, very smart and have efficient software gates built in.

Local Anaesthesia (Reducing The Pain)

For the new CDTV developer, the prospect of a read-only test-and-cry medium is daunting. The good news is, it stays daunting. The early history of CDTV is littered with very expensive and totally useless CDs.

The only way to keep the terror down to manageable proportions is a sound methodology. If you are going to make a mixed-mode disc, you will need an audio proof disc at the earliest possible stage. Trying to cut this stage out of the loop will cost more, not less. Get the audio finalized, cleared for world copyright, recorded and then onto a CD. If you can get all the data onto the CD at the same time, do it because it makes life easier. If the audio is on and the data is on, you can hone your program(s) with nothing more than a floppy drive plugged into the system. This is the preferred method, though if you cannot build the whole thing at this stage, you may have to use a SCSI board in the CDTV expansion port and a SCSI device hung onto that.

As mentioned before, when you lay your audio, allow for the data track, even if it means that track ONE lasts fifty minutes and contains nothing but silence.

Use a good CD manufacturer. The good companies know everything and more about the physics of the disc, the ISO 9660 standard, audio balancing, the works. The bad ones have a Yamaha One-Shot machine and lots of potted plants.

As with all other CDTV discs, the worst element after testing, software finalization, etc., is making sure that the startup-sequence and preferences settings are correct. This is very important after working from SCSI or floppy boot because you will no longer be using the startup-sequence on the floppy or SCSI which may have different assigns on it.

During development, you may want to address files on the floppy or the SCSI rather than the CD. One method is to use paths to logical devices in the code—for example CD99:. Your startup-sequence on the floppy or SCSI will assign that to itself most probably...but in the final system you simply assign CD99: to CD0: in the startup-sequence. This really does make life easier.

Last Thoughts

The *cdtv.device* is powerful. It does nearly all the work for you. The biggest rule and the most important is make sure your file control is 100% and make sure the audio is stopped and cleaned up before continuing, remembering that interrupt calls from VBeam or FrameCall may need some time to disable. It's also a good idea to make sure that you can never get into the situation where you try to access a picture file or something like that while the audio is running. This is a guaranteed way to lock up the system.

With the above things in mind, the system becomes easy and nice to use. It's a real pleasure to hear a high quality CD-DA track come up and play under your program control, and all you need is a few lines of code.

You can end up swapping CD discs often during development. If you start to get increasing errors it almost certainly means your CD is dirty. Take it into the bathroom, wash it with soapy water, dry it with a towel, and shove it back in.

Try doing that with a floppy!



CD-DA Sound

Recording 16-bit Sound

The CD-DA standard allows for high quality, flat response and very low noise playback. To get the most from it requires a professional approach to production, recording and engineering. Computer users are often so impressed with the fact that there is any sound at all that they will accept noisy, hissy sub-telephone quality. Compact Disc users expect clear, noiseless, radiant sound, and will not be sympathetic to anything else.

Many CDTV publishers will go to professional recording studios to have their audio captured and edited ready for mastering. Apart from considerations of sound quality, the most important factor for developers when relating to the studio will be the way the final track or tracks are presented for mastering. The options involve a decision about the way the CD-ROM functions under program control.

If the title breaks naturally into sections of audio—e.g., a Karaoke disc with fifteen music tracks—the audio layout will reflect that. Track 1 will be the data track with pictures, words and controlling programs, and tracks 2 to 16 will be the audio tracks. An easy trap to fall into there involves track 1. Studios will not normally be used to having a data track as track 1 and will produce the audio tracks from 1 upwards. This may cause confusion later about the numbering of the audio tracks. It's a good idea to have the studio make track 1 on the DAT or tape it produces be thirty seconds of 0dB tone. This helps the transfer to mastering equipment and retains the same numeric layout of tracks.

A second potential problem is the track lead-in silence. Tastes differ between studios on this. Some will allow a few seconds of silence at the track start, others will electronically trigger the DAT marker so that the actual audio starts on logical frame 1 of the track. Either way, consistency is desirable if you will later be synchronizing other events to the track.

A third problem is the mastering house. It is important to check before going to a studio the audio formats supported by the mastering house. Some will accept reel-to-reel tape, some will accept 12 or 24 track masters, some will accept audio DAT, and some will be able to make a digital copy of a DAT tape. In the last case, bear in mind that DAT can run at two rates: approximately 48,000 samples/second and approximately 44,100 samples/second. If you are hoping to use digital transfer it is vital that the studio use 44,100 on the DAT, because that is the CD sampling rate.

An alternative approach to track layout is where your title uses hundreds of small segments of audio data—e.g., historical speech clips from world figures. It may be better for this to use only one audio track which can be addressed by the program in time fragments—the MSF (Minute Second Frame) standard. This allows tiny subdivisions of the audio track. If fades in and out are required on short clips, it may be advisable to have these engineered at the recording stage, otherwise the controlling program will have to retain fade time and position information for each segment. If the MSF approach to track layout is used, some effort in programming can be saved by using a recording studio which is equipped with a good direct-to-disc digital recording system. This will

allow the engineer to mark *cue points* for each of your small audio segments, and print these out for you. These cues will give you the MSF information you need for correctly launching the audio data.

Where a lot of shuttling backwards and forwards between audio sections may be required by the program, it is useful to arrange the audio segments so that those most likely to be played in sequence are as close together in the track as possible. The seek time on the CD may cause delays of up to 0.8 seconds between widely spaced audio. These times are very difficult to compute accurately without a proof-disc, and will be affected by intervening calls to the CD-ROM track to read data, which will reposition the laser near the disc hub.

Audio Quality Guidelines

Getting the best from a recording studio is beyond the scope of this section. But if you have to record sound away from a studio, or on a "Do It Yourself" basis, some simple techniques will help the final quality:

Speech and Interviews

The best position for a microphone recording speech is nine to twelve inches from the speaker's mouth, positioned below it. Do not let the speaker speak *into* the microphone. This causes increased sibilance and popping as the airstream from the speaker hits the microphone. Too close is as bad as too far away. As the distance increases, the proportion of ambient sound in the recording rises, and the speaker appears to the listener to be much more distant she really is. The ranges used by television interviewers are *not* a good guide because the picture provides vital spatial information which modifies the spatial data in the sound alone.

Recordings should not be made in bare rooms. The echo massively overemphasizes the perceived distance of the speaker. If an interview must be done in this kind of environment, place a chair on a table and hang a coat over the chair. The interviewee and you should get as close as possible to the coat, however bizarre and intimate a situation this creates. Microphone cables cause very nasty rumbles and clicks. The correct technique is to wrap some cable from the recorder around the hand, so that the hand takes up any cable rustle. The microphone should not be held lightly like a chopstick, but gripped with as much pressure as can comfortably be exerted.

Recording levels should be adjusted to be just below peak distortion level. Low level recordings are hissy, but it is worth remembering that most people *increase* their volume once an interview starts, so a careful eye should be kept on the level meter if the recorder is not in automatic record mode. (Automatic recording should be switched off in noisy environments. It can cause "hunting", which is unpleasant to listen to.)

Classical Music

Orchestras, chamber orchestras, etc., are best recorded with a stereo microphone pair about six feet behind the conductor and at a height of eight to ten feet. Putting spot microphones into orchestras is a very advanced recording skill, and best avoided unless you have a complete mobile sound studio. Unlike folk and rock groups, orchestras, string quartets and other essentially "acoustic" ensembles are best recorded in nice lively acoustics such as wood paneled rooms. Adding electronic reverb to a good orchestra is like retouching a Rembrandt.

Acoustic Folk

To get the best results, it is necessary to supply a microphone for each instrument and voice. Acoustic guitars are best captured by pointing a microphone at the wood between the hole and the end opposite the screws. In general, you need to get as much separation between all the elements as possible and rebalance them on the mixer.

Rock And Electric Sounds

Either take an audio feed directly from the instrument, or place speakers in the studio sufficiently far apart so that a microphone approximately twelve inches from the speaker will get most of its input from that speaker and not all of them. Put the drum kit well away from everything else. For the best results, bully the band into turning the equipment down to the lowest level that will enable them to play. Add plenty of reverb. The worse the band, the more reverb required.

Drum Kits

Recording engineers will argue for hours about the best microphone to use for each element of the drum kit. For acceptable recordings, use one microphone positioned above the upper part of the kit (snare, cymbal, etc.) and a separate microphone for the bass drum. The bass drum should have its front removed and a pillow or cushion stuffed into it.

Recording In The Open Air

If you are trying to capture an acoustic band (wind band, etc.) playing in the open air, persuade them to play near a tree or other high but accessible object, and place the microphone as nearly above them as possible. In the open air, high frequency sound goes up like pure hydrogen, and the closer you are to its path, the better.

Piano

Upright pianos should have either a microphone (or two) placed above the open lid, or beside the feet of the pianist. Grand pianos should have at least one microphone inside the open lid pointing down at the strings, or below the piano about twelve inches from the floor pointing upwards.

Careful positioning with even a cheap mixer and cheap microphones will bring better results than bad positioning with high quality equipment.

CD-DA has very low noise, even at low recording levels. There is no need to compress sound and force it up to a high average amplitude unless that is the desired effect. The complete dynamic range may be used. This is quite different from the 8-bit recording technique, which usually requires a high average amplitude.

CD-DA Plus 8-Bit Audio

There is no simple equation that will tell you what the relative volume settings should be when playing a 8-bit Amiga sound sample at the same time as CD-DA. The ratio is a function of the final sound energy of both waveforms. The only way to establish the correct balance with any certainty is to cut a proof-disc and adjust the volumes either by altering the 8-bit volume, or the CD-DA volume. The "density" of the sound may be as important as the actual volume.

If both sound generation systems are to be used together, it is very important that the CD-DA volume not change between cycles of the pre-mastering process. If the pre-mastering house has to redigitize the sound tracks, there is no guarantee that its equipment will operate at the same input

level without a clear 0-level reference tone being placed on the audio tape (and even then it is not always guaranteed). CD-DA tracks should be finalized, recorded, transferred to the pre-mastering equipment and stored in digital format at the pre-mastering house unless changes have to be made to them.

Playing CD-DA Tracks

The *cdtv.device* makes playing CD-DA tracks very simple. Two main play mechanisms are provided—by track or by time offset. In both cases, the play can be synchronous or asynchronous, as required by the programmer.

The *cdtv.device* accepts messages in *StdIOReq* format passed with either *DoIOO* or *SendIOO*. The relevant constants and structures are defined in *cdtv.h*.

The following code examples assume (for brevity) that two *MessagePorts* named *Port1* and *Port2* have been successfully opened, that two *IOStdReq* structures have been allocated with pointers to them in *pIOSRQ1* and *pIOSRQ2*, and that these two requests have successfully performed an *OpenDevice()* on *cdtv.device*. The examples are in HiSpeed Pascal™, but are simple to transfer into C, Modula-2, assembler, etc.

```
{Example function to perform a synchronous track play}

Function PlayTrack(track:integer):boolean;
VAR x:Integer; { for DoIO return code }

begin
  IF ((track<1) or (track>99)) then { illegal track number }
  begin
    PlayTrack:=false;           { return error      }
    exit;
  end
  else
  begin
    With pIOSRQ1 DO
      begin
        IO_Command:=CDTV_PLAYTRACK; { play by track number }
        IO_Offset :=track;          { no guarantee this track actually
                                     exists, but try      }
        IO_Data:=NIL;               { zero unused fields  }
        IO_Length:=0;
      end;
      x:=DoIO(pIoRequest(ISRQ1));    { with                }
      PlayTrack:=(x=0);              { Do request          }
    end;                             { return the success of the DoIO }
  end;                               { if/else              }
end;                                { PlayTrack            }
```

Assuming the track exists (and there is a CD present!), this call will play the CD-DA track and return when the track concludes. Of course, this function could be extended to return the actual error returned in the *IOStdReq* message if an error occurred.

The asynchronous version of the above code sends the message and then returns, leaving you to perform screen magic or Midi I/O, etc., while the audio is playing.

```
{Example function to perform an asynchronous track play}

Function LaunchTrack(track:integer):boolean;

begin
  IF ((track<1) or (track>99)) then { illegal track number }
  begin
```

```

LaunchTrack:=false;           { return error           }
exit;
end
else
begin
  With pIOSRQ1 DO             { set up StdIOReq       }
  begin
    IO_Command:=CDTV_PLAYTRACK; { play by track number }
    IO_Offset :=track;         { no guarantee this track actually
                                exists, but try         }
    IO_Data:=NIL;              { zero unused fields   }
    IO_Length:=0;
  end;
  SendIO(pIoRequest (ISRQ1));  { with                 }
  LaunchTrack:=TRUE;          { send request         }
end;                          { assume success of the SendIO }
                              { if/else                 }
                              { LaunchTrack                }
end;

```

The track should now be playing. A short pause to allow it to start, followed by a call (with a separate `IOStdReq`) with the commands `CDTV_STATUS` or `CDTV_QUICKSTATUS` should establish that the track is there and running. With your own CD this should not be necessary once the program has been debugged because you will know how many tracks you have. Where users are invited to insert their own discs, it is wise to read the disc's Table Of Contents (TOC) to find out how many tracks there are, and be sure never to play track 1 if it is a ROM track.

Assuming the audio track is playing, the next concerns that arise are:

Has it stopped?

How Do I Stop It?

The `CheckIO()` system call returns the status of a pending I/O Request, so:

```
over:=CheckIO(pIoRequest (pISRQ1));
```

When `over` becomes `TRUE`, the track has stopped. Whether it has, or whether we want to stop it, the following code is used:

```

AbortIO(pIoRequest (pISRQ1)); { abort the request }
er:=WaitIO(pIoRequest (pISRQ1)); { wait for abort   }

```

The `cdtv.device` does not complain if terminated I/O Requests are aborted, and this makes the code simple and general. But, although the I/O has stopped, asynchronous play requests do not normally switch off the drive light and restore the system. The following additional code should be added:

```

pIOSRQ1^.IO_Command:=CDTV_STOPPLAY;
pIOSRQ1^.IO_Offset :=0;
pIOSRQ1^.IO_Data:=NIL;
pIOSRQ1^.IO_Length:=0;
er:=DoIO(pIoRequest (pIOSRQ1));

```

The `CDTV_STOPPLAY` should ensure that the system is now completely stable and safe for file transfers. The command may be issued with a different `IORequest` before the `AbortIO()`, but the required code will be essentially the same.

Using PLAYMSF

The code examples in the previous section deal with complete CD-DA tracks on the CD. In many cases it may be desirable to play a fragment from a track, and this can be accomplished with the same ease as playing a track. Two formats are provided: CDTV-PLAYLSN and CDTV_PLAYMSF. The PLAYLSN command requires detailed knowledge of the disc geometry and will only be used under exceptional circumstances, so we concentrate only on the MSF format.

The Minute, Second and Frame values are UBYTE numbers packed into a ULONG with a Reserved byte in the "leftmost" byte. The *cdtv.h* header file defines a union type which allows easily manipulation of the interior data in the various ways the device driver expects to see it.

MSF Has 75 Frames. Although the MSF format corresponds to SMPTE terminology, there are 75 CD frames/second, rather than the 24/25/30 used in SMPTE handlers. The range of the Frame byte in the CDTV RMSF format is therefore 0-74.

A C Macro to pack the fields is provided in *cdtv.h*.

The only difference between using CDTV_PLAYTRACK and CDTV_PLAYMSF is that the CDTV_PLAYMSF command expects two fields to be filled in with the packed RMSF long words. The Offset field contains the starting time and the Length field contains the stopping time. The behavior is otherwise exactly as above, and whether it is synchronous or not depends on whether you dispatch the request with DoIO() or SendIO().

Reading The Table Of Contents

The TOC (Table Of Contents) of a CD lies outside the ISO-9660 ROM image, and is present for CD-DA only discs as well as CD-ROM discs. It is not exactly a King Solomon's Mine of Information, and will not, unfortunately, tell you what a track is called, but only its start time, and what kind of track it is.

In Pascal each entry for a track is defined as follows:

```
TYPE CDTOC=RECORD
    rsvd      : BYTE;
    AddrCtrl  : BYTE;
    Track     : BYTE;
    LastTrack : BYTE;
    Position  : tCDPOS; { packed R M S F }
END;
```

The AddrCtrl field contains various bit flags indicating the track type. The exact interpretation of the other fields depends on which track we are reading because although there is no "real" track 0 on the CD, the TOC itself constitutes a phantom track 0. For the 0 track entry, LastTrack contains the number of the last track on the CD, which also, as it happens, is the number of tracks. This field is *not* valid for any other entries.

The *cdtv.device* command CDTV_TOCMSF allows you to read up to one hundred entries into an array of the CDTOC structures. The Offset field in the IOStdReq structure determines that track to start at, and the Length field determines the number to read. Obviously, you can determine the number of tracks by reading one track from *Offset=0* and then allocate a dynamic array to read only that number of tracks. In the following example, a static array big enough to hold the entire TOC

is declared, and the procedure `GetTOC` gets all the disc data into the array. Elements above the `LastTrack` value in element 0 of the array will be invalid.

```
VAR DiskTOC:array[0..99] of CDTOC;

{ Pascal procedure to read entire TOC from CD }

PROCEDURE GetTOC;
VAR doIOresult:integer;

begin
  With pIOSRQ1 DO
  begin
    IO_Command:=CDTV_TOCMSF; { data in Position is R M S F format }
    IO_Offset :=0;           { start from track 0 }
    IO_Data:=@DiskTOC;       { address of array }
    IO_Length:=100;          { read entire toc }
  end;
  DoIOresult:=DoIO(pIoRequest(pIOSRQ1)); { do it! }

end; { GetTOC }
```

If all went well, the TOC data will now be in the array `DiskTOC`, but the tracks only have their *start* times in the TOC entry. To find the end of a track, use the value in the following entry and subtract at least one frame. Well—that's fine for all tracks but the last one, which does not have a successor! To find the end of the last track, use the value in the `Position` field of `DiskTOC[0]`. This is the start of the lead-out area on the disc.

It is normal to lay out discs with two seconds of digital silence between tracks. *This cannot be relied upon.* Some Classical CDs have tracks that follow with no break at all.

The simplest way to determine whether a disc is mixed-mode, CD-DA only, or simply an ISO-9660 disc with no audio, is to use the `CDTV_ISROM` device call, which is handled much as the examples above. This call returns a boolean value in the `io_Actual` field of the `StdIOReq`. For most disc cases, this will be adequate, but if you want to have more detail, examine the flag values in the `AddCtrl` field in conjunction with *cdtv.h*.

Timing Considerations

For close synchronization with CD-DA tracks, the following considerations apply:

1. The audio tracks must be in *exactly* the same place for each CD.
2. The start of play must be checked until it stabilizes.
3. A clock must be running that is locked to the CD-DA.

The best way to achieve absolute control on the CD-DA tracks is to have them placed in position only once, to have a proof-disc with the tracks on, and to ensure that this data is stored and reused by the mastering house. Accuracy to one frame is usually impossible; five frames is normal. Bear in mind that neither CD players nor DAT recorders are absolutely accurate. It is possible to compound speed errors to achieve a significant mismatch between tracks by making assumptions about accuracy.

In practical terms this means assign enough room for track 1 to allow for all your possible changes to code and data. You can then ask for the CD-DA tracks to start at an offset that equals, for example, 100 Mbytes. The tracks are sampled or transferred to the control system for proof-disc and master disc cutter. For each iteration of the development process, your data track image is substituted for

the track 1 space on the control system. You can then be certain that the audio tracks are *identical* in format and position to those for the last iteration.

Checking for exact start of play is a run-time function. Although you may send an `IORequest` asking for an asynchronous track or MSF play, you have no means of knowing whether this will take 0.5 seconds or no time at all. The easiest way to check that your track has started audio play is to use the `cdtv.device` `CDTV_QUICKSTATUS` call. As you have sent your play request on the primary `IOStdReq` structure, you must have another ready to do this. `CDTV_QUICKSTATUS` returns a number of flags in the `io_Actual` field. Writing complex code to examine these bits minutely can cause timing imprecision. When the track is valid and audio play is happening, `CDTV_QUICKSTATUS` returns a value of 101 decimal. (If an error occurred, it will *never* return this, so your check loop should allow for an error count-out or time-out.)

```
The launching algorithm is evidently: Set up track or MSF play
SendIO() the request
REPEAT UNTIL QuickStatusResult=101 OR TimeOut
```

Amiga purists will probably complain that this implies a busy-looping call to `DoIO()` with a `CDTV_QUICKSTATUS` command, which is exactly what it is. It does busy-loop, and it should. For tight synchronization, other tasks should be ignored.

At this point, your track is playing or a major error occurred. Once you have a proof-disc and have debugged silly track numbers or MSF values out, the call should be reliable. The question then is: How do we synchronize events to it?

The simplest and most efficient synchronization is achieved by installing some code to be handled on the CD frame interrupts, which happen seventy-five times per second. There is no need to set up an `Interrupt` structure, because the `cdtv.device` handles the low-level aspects of this for you. However, it does expect you to get into and out of your code quickly. The Frame code is executed in Supervisor mode. It will not permit long and fancy tricks. Typical frame synchronization code will be:

```
{ Global Clock variable has been declared as type Longint }

Procedure AddClock;
begin
  INC(Clock);
end;
```

This unimpressive routine is added to the system by a call to the `cdtv.device` with the `CDTV_FRAMECALL` command. When this code is posted to the `cdtv.device`, the `Clock` variable will increment seventy-five times per second. You will need a separate `IOStdReq` for *each* Framecall code block you add (you can add as many as you like) because they are requested with `SendIO()` and stay there until an `AbortIO()` referring to them is sent.

The `AddClock` code is locked to the system like this:

```
With pIOSRQ2^ DO
begin
  IO_Command:=CDTV_FRAMECALL; { request addition of code      }
  IO_Offset :=0;
  IO_Data   :=@AddClock;      { address of framecall code    }
  IO_Length :=0;
end;
SendIO(pIoRequest(pIOSRQ2)); { added to system              }
```

At the point that our `QuickStatus` checking above returns successfully, we can now do:

```
Clock:=0;
```

The **Clock** will now increment seventy-five times per second, and we know that this is very tightly synchronized with the track play. **Clock** will continue to increment until the **IORequest** is aborted.

This looks too good to be true, but in fact, it is that easy. The important point to remember is that each **Framecall** requires its own **IORequest** because it stays on the device queue until you **AbortIO()** it. Trying to reuse **IORequests** which have been sent to the device but not returned is *eventually going to cause grief*.

As always, the danger with asynchronous I/O on a CD is in assuming that it is safe to do a file transfer without being absolutely certain that the track play has terminated. This is the most common source of difficulty in CDTV development. It is good software design to encapsulate all I/O to the CD within one module, with guards against any track play during file I/O or file I/O during track play. With *cdtv.device* calls spread across a range of code blocks, the chances of I/O contention rapidly increase.

PAL And NTSC

The DMA clock rates for PAL and NTSC machines are different. The audio pitch for a sampled sound played at the same audio DMA rate on each system will differ by a fraction of a semitone. For speech, the difference is insignificant. For 8-bit Amiga SMUS track play, the difference is also so small as to be insignificant. But where pitched notes generated on the 8-bit Amiga hardware are merged in the audio with CD-DA tracks, the difference may be painful. The CD-DA pitch is *not* affected by PAL/NTSC considerations.

With Kickstart 1.3 running on PAL CDTVs or Amigas, there is a chance that the system will incorrectly boot into NTSC mode. This problem has been solved under Kickstart 2.0. Assuming that you know which machine your code is running on, it is important to use the correct divisor to compute the audio DMA ticks for a sample at a given frequency.

For frequency **Freq**, the audio DMA ticks are calculated as:

```
NTSC ticks = Freq DIV 3579545;  
Pal  ticks = Freq DIV 3546895;
```

For finely-tuned music titles, it is unsafe to assume that even this strictness is enough. Calculations of note frequencies within high octaves may need "hand" adjustment for the two systems.

Coda

The use of CD technology has two benefits: storage and sound quality. If the requirements of storage allow for it, there is no contest between 8-bit sound and full CD-DA quality. Playing CD-DA tracks on CDTV is easy. Unless the title must load data from the CD at the same time as sound is playing, CD-DA is the audio format to use.



MIDI

Overview

The purpose of this article is to explain MIDI and how to use MIDI in a CDTV application.

Specifically, this article covers:

- MIDI Overview
- CDTV & MIDI—possible applications
- Available Tools
- Programming Techniques
- CD & MIDI
- Tools Resource List

MIDI Overview

WHAT IS MIDI?

MIDI is an acronym that stands for *Musical Instrument Digital Interface*. The acronym's meaning makes MIDI sound like a noun, but it is much more. It is not only a type of interface for connecting musical instruments and other equipment together, MIDI is also a complete protocol specification for how MIDI-equipped devices can talk to each other.

In effect, MIDI is a local area network for musicians. It allows devices to be connected together so that they can communicate. Therefore, equally valid questions are "Does that synthesizer keyboard have MIDI?"—meaning does it have a MIDI interface (connectors) built-in—and "What are its MIDI capabilities?" which asks how flexibly the keyboard can be controlled or control other equipment.

THE PROBLEM THAT CAUSED THE DEVELOPMENT OF MIDI

Electronic musical instruments—usually referred to as *synthesizers*—are sophisticated sound generating devices with controls such as keys, wheels, pedals, buttons and knobs to shape the resulting sound. There is a variety of synthesizer types, each creating distinctive types of sounds. Synthesizers usually fall into one or more of the following broad categories: Analog, Digital, Sampler and FM. These categories describe the methods the synthesizers use to generate sound. The details of those methods are outside the scope of this article.

Synthesizers, Synthesizers, Everywhere Synthesizers

Most professional synthesizer players have multiple synthesizers from the different categories mentioned, and it became clear in the very early days of synthesizer development that it was very cumbersome to have to configure and use all these different types of synthesizers, particularly live. You may have seen rock concerts from the 1970s where a keyboard player was surrounded by a dozen keyboards. It was basically a nightmare to keep all of this equipment configured properly, not to mention physically demanding for the musician to have to turn around to find the keyboard he wanted to play at any given point in the performance.

In the late 1970s and early 1980s, great minds got together and decided that a way to connect synthesizers together was necessary to simplify using them. Under such a system, a musician could use one keyboard not only to play its sounds, but also to control and play other keyboards and devices. This would mean the end of keyboard players spinning around on stage among rows and rows of keyboards arranged in a circle. Shortly thereafter, MIDI became a reality.

PHYSICAL DESCRIPTION

Physically, MIDI is very simple. The interface is basically a fast (31,250 baud, to be exact) serial (RS232) port with receive and transmit signals connected to separate round, 5 pin 'DIN'-type connectors specified as IN and OUT respectively. If you look at any MIDI-equipped device (usually the back,) you will see one or more of these 5 pin DIN connectors.

Some devices do not have any need to transmit data, and therefore do not have an OUT (transmit) connector. An example is a MIDI-controllable light box that can be sent MIDI data to turn lights on and off. There is no reason for this light box to ever send data to anything else, so it doesn't have a MIDI OUT connector.

Other devices may not need to receive any data, and therefore don't have an IN (receive) connector. An example is a drum pad that sends MIDI data out whenever someone hits it with a stick.

Connecting the output of one MIDI device to the input of another opens up many interesting and useful applications. If a MIDI cable is connected from the MIDI OUT connector of the hypothetical drum pad to the IN of the hypothetical light box, then these devices could be configured such that whenever the drum pad is hit, a light could go on.

SOUND MODULES

Once the door was opened for one keyboard to control one or more others, it became clear that the *keyboards* being controlled didn't actually need a physical keyboard, which is usually the most expensive part of a synthesizer. All these other keyboards were being used for was their sound generating capabilities.

So a whole new generation of synthesizers now exists, which are usually referred to as *synthesizer modules*. These modules incorporate synthesizer sound-generating electronics, without the keyboard, and can be mounted in a rack. They are connected—via MIDI cables—to other devices, usually a keyboard, which send data to them to cause them to produce sounds.

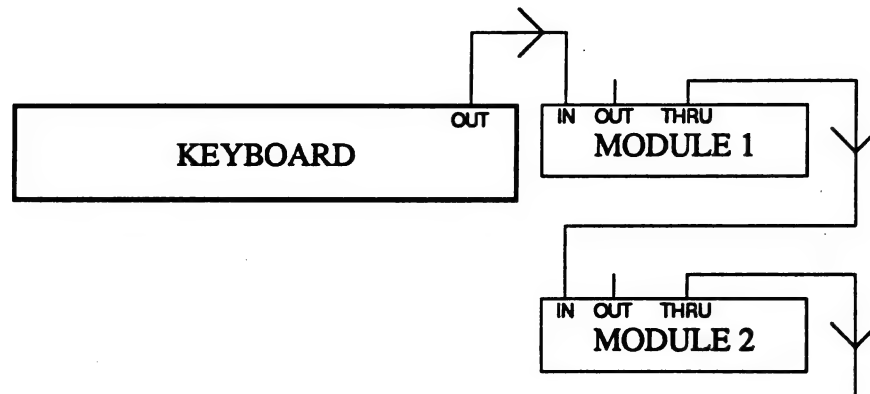
MIDI THRU

If the communication between synthesizers consists of a transmit and receive, you may be wondering how one keyboard could talk to more than one other keyboard or sound module. The answer is called *MIDI THRU*, and physically consists of an additional MIDI connector on many devices usually marked *THRU*. This connector automatically outputs whatever data arrives at the MIDI IN connector. Therefore, one keyboard can communicate with two sound modules, for example.

MIDI DATA FLOW

When typically configured, a key pressed on the keyboard in the above diagram will cause MIDI data to come out of the MIDI OUT connector, and flow to the IN connector of module 1. It will automatically be *mirrored* at module 1's THRU connector, and flow to module 2's IN.

The second module's THRU connector could be connected to the IN of something else, and so on.



MIDI CHANNELS

The question arises, if I play the keyboard in the above diagram, does it mean that a sound will come out of the keyboard and both sound modules? Clearly if that were the case, MIDI's usefulness would be greatly diminished.

The MIDI specification allows data to be organized into *channels*, up to 16 of them running through one MIDI cable, and MIDI devices can be configured to respond to one, many, all, or no MIDI data channels. There is only data transmission wire in a MIDI cable and not 16, so the data bytes themselves travelling along that one wire contain information specifying which channel the data is associated with.

If the data coming from the keyboard were specified as channel 1, and module 1 is configured to make sounds when it receives notes on channel 1 while module 2 is configured for channel 2, then only the first of the two modules would make sound when keys are hit. If the keyboard is then configured to transmit on channel 2, the first module will stop making sound, and the second will start.

Just to add to the confusion, it should be noted that one sound module or keyboard might respond to more than one channel, in effect, acting like multiple sound modules. These are called *multitimbral*

devices. Sound modules or keyboards can usually be configured to respond to *any* channel also, which is known as *OMNI* mode.

And to totally confuse you, more than one sound module may respond to the *same* channel, allowing someone to play multiple sounds at once with a single key press.

This flexibility has made MIDI a useful standard that has withstood the test of time and progress.

LOCAL CONTROL AND CONTROLLERS

But, you may be asking, isn't the keyboard still making sound also, in addition to one of the other modules? The answer is: if you want it to. Most keyboard synthesizers can be configured as to whether pressing the keys will play the internal sound electronics or not. This is called *local control*.

Think of a keyboard synthesizer as two parts: a keyboard, and a sound module. The two components just happen to be in the same physical case. The physical keys on the keyboard is a *controller*, i.e., something that the player uses to control the music. Other examples of controllers include modulation and pitch bend wheels (common on most synthesizers), foot pedals, breath controllers (which, as their name implies, you blow into, and they generate MIDI or electrical control information), and drum pads which don't sound like drums when you hit them, they just generate MIDI information that can cause a sound module to make sound.

Specifically, local control determines whether the controllers in a synthesizer keyboard automatically cause the built-in sound module to generate sound or not. On some synthesizers, local control can be specified for each controller, e.g., the keys may make sound locally (as well as generating MIDI data, if desired), while the wheels might not. An example of how this control might be used is a foot pedal connected to a keyboard—it may not do anything to the keyboard, but the MIDI it generates will do something to another sound module.

MIDI DATA

MIDI data appears to any MIDI device as a stream of one or more bytes. Most bytes are part of a multiple byte message or *event*. For example, pressing a key on a keyboard generates 3 bytes of data. The first byte indicates that the event is a *NOTE ON* event, and also specifies the channel. The second byte indicates which key was pressed, as a value from 0–127. The third byte indicates how fast the key was hit, which is known as *velocity*. The velocity of a note might change the timbre of the resulting sound, as is the case with traditional instruments such as a piano.

Other types of *events* or data packets include:

- the activity of another controller such as a wheel or pedal
- pressing harder on a key while it is depressed (known as *aftertouch*)
- information specific to a particular piece of equipment (known as *system exclusive* which is ignored by equipment that does not recognize the header of this information which specifies the manufacturer, and data stream type).

MIDI SUMMARY

To recap, and to repeat, MIDI is like a local area network designed for musicians. It allows equipment to be connected together so that they can communicate and control one another.

CDTV & MIDI—Possible Applications

CDTV is a MIDI-equipped device, i.e., it has MIDI IN and OUT ports on the back. It can serve as both a controller and a sound module through software and the built-in, 4 voice 8 bit digital audio, respectively.

HOW CAN MIDI BE UTILIZED ON CDTV?

Musical applications can basically be broken down into two categories: consumer and professional.

Consumer

Consumer music applications can either be the *sit back and watch* variety, or better, interactive ones that give users the feeling that they're as talented as Mozart. The sit back and watch type of application might use the CDTV as a jukebox, pumping out previously created MIDI sequences to whatever synthesizer the CDTV is connected to. The interactive type of application should let users have fun by interpreting their interactions in harmonically interesting ways. Perhaps the program can improvise accompaniment to go along with a user's playing. An outstanding example of this type of program is called *Band-In-A-Box*, although it is not available for the Amiga yet, unfortunately.

Professional

Professional applications (or more specifically, ones for people interested in creating music) should provide the power to create, edit and store musical compositions or sounds. Perhaps a rack-mounted CDTV with professional sequencing software could be useful for musicians because of the compact nature of the device, and the way in which it would integrate in a rack with other equipment.

MIDI sound libraries (parameter sets) for every known synthesizer could be sold on one CD, with a nice simple-to-use interface.

Available Tools

There are many programs available that 'do' MIDI. Some do MIDI only, while others handle MIDI in the context of other features, such as a multimedia authoring program that can play animations while outputting MIDI data.

SEQUENCERS

Sequencer: a computer program or device that can capture, play back, and manipulate MIDI and/or other (usually music-related) data. In its simplest form, a sequencer is like a multitrack recorder, but the more powerful ones are like the musical equivalent of a word processor.

Many Amiga-compatible sequencer programs should run unmodified on the CDTV if a keyboard and floppy disk drive are connected. These sequencer programs allow you to create MIDI and/or CDTV digital audio (not CD audio) compositions. The audio output of the CDTV and/or synthesizers could then be recorded and pressed onto CD audio tracks for use in your application, but that process is outside of the scope of this article.

To record musical compositions, you should connect one or more synthesizers to the CDTV unit, run the sequencer program, and record one or more *tracks* of notes/events. Sequencers can play back previously recorded tracks as you record additional tracks. Typically, each track would be set to a different channel so as to play a different sound module or keyboard. As mentioned above, a modern, single sound module or keyboard that can respond to multiple channels, playing different instruments on each channel are termed *multitimbral*. One very popular (but no longer sold,) small, desktop-sized multitimbral sound module is the *MT-32* from Roland. It can respond to 8 channels at once, allowing for full compositional arrangements to be made. The *MT-32* also incorporates a simple digital reverb which provides real world acoustics instead of a dry, dead sound.

MULTIMEDIA TOOLS

AmigaVision 1.8 will support output of Standard MIDI Files, which will (combined with all of its other powerful features) make it an ideal tool for CDTV authoring.

ShowMaker from Gold Disk supports time-line editing of MIDI tracks, Amiga graphics, and other events, to allow the creation of spectacular multimedia presentations.

Director 2, a powerful authoring system from the Right Answers Group, also supports MIDI.

Other multimedia tools may offer MIDI capabilities also, and should be investigated.

Programming Techniques

The analogy of MIDI to networking is quite accurate, and can involve a tremendous amount of tricky programming. Because MIDI data is used for musical applications, there are also a large number of very tricky timing-related issues. The ear—and subsequently, the brain—is *very* sensitive to music tempo changes, which means that the timing of musical MIDI data input and output must be accurately tracked.

Rolling Your Own MIDI Routines

Creating your own MIDI routines is not an easy thing to do. It requires extensive knowledge of the CIA timers, the serial port hardware, and efficient interrupt coding. This article would be enormous if all of these issues were covered.

It should be noted that the Amiga OS 1.3 *serial.device* is too slow to handle MIDI data efficiently, particularly considering the fact that the CDTV is only a 68000 based system, and that the MIDI baud rate is a relatively fast 31,250 baud.

CAMD

Fortunately, CATS will offer a tool called the *Commodore Amiga MIDI Driver*, or *CAMD* for short. This driver handles all of the low level I/O and timing issues on a CDTV or Amiga system, providing a higher level interface for your application.

CAMD is an Amiga OS run-time library, and is opened, used, and closed just like any other system library such as *graphics.library* and *intuition.library*.

CAMD History

CAMD was originally developed at Carnegie-Mellon University. It was then improved and optimized by Bill Barton, and has since been improved and tested inside Commodore to meet in-house standards.

Time-stamping and Filtering

As was mentioned above in the "Programming Techniques" section, timing is critical in musical MIDI applications, and *CAMD* handles the time-stamping of incoming data automatically. That is, the time at which each incoming MIDI message is received is stored with that message.

Also, your application can tell *CAMD* to ignore certain kinds of data, allowing your code to handle only what it is interested in. For example, you may not care if users of your application are playing around with pedals and wheels on their synthesizers (which can generate a lot of data), but you need to know if they're pressing keys on the keyboard. *CAMD* will allow for this kind of filtering.

Virtual Port Management

CAMD will usually be used to process data coming into and going out of the standard MIDI ports (which are, in fact, just different connectors on the serial interface). However, it can also read data from and write data to 'virtual' MIDI ports which don't have any physical hardware attached, but might be inputs and outputs of other running applications. This facility can be used to allow MIDI application modules to have an input and an output that could be either the physical ports, or the output and input from another running module. This allows for a modular approach to MIDI software, which is ideal in a multitasking environment.



CTrac Emulation System

Introduction

The *CTrac* CD-ROM emulation system, developed by ICOM Simulations, Inc., is a combination of hardware and software that emulates CD-ROM discs for the CDTV. The emulation system creates an image of a CD-ROM disc in a file on the hard disk drive of an Amiga. The system then acts as the CD-ROM drive by presenting the image to the CDTV, as if it were an actual CD-ROM disc. Since the emulation system replaces the CD-ROM drive in the CDTV, all formats and modes of CD-ROM can be emulated including CD-DA, CD+G and CD+MIDI.

This is a true emulation environment, not a simulation. The application actually runs in the memory of the CDTV. The emulator board replaces the CD-ROM mechanism. Every time the CDTV tries to access the CD-ROM, it accesses the emulator board. The board and the emulation software in turn access an ISO 9660 disc image on the Amiga's SCSI disk drive. The board slows down the seek times of the SCSI drive, as well as the data transfer rate, to accurately emulate the performance of a CD-ROM drive.

Until now, the only accurate way to test CD-ROM applications was to have the application pressed on to a CD-ROM disc and then tested in the CDTV. The process of pressing a disc to test an application under development is time consuming and costly, especially if the application does not work properly. Emulating CD-ROM applications with the *CTrac* emulation system not only eliminates the need to have discs pressed for testing but also allows accurate monitoring of the commands sent to the CD-ROM drive. We do recommend that a test disc be cut before mastering the application, however.

The *CTrac* CD-ROM emulation system hardware consists of a printed circuit board that plugs into an Amiga 2500 or Amiga 3000, and an interface cable. The cable connects the emulator board to the CD-ROM drive connector in the CDTV. The *CTrac* hardware together with the *CTrac* software running on the Amiga emulates the CD-ROM drive in the CDTV. The emulation is completely transparent to the CDTV.

CTrac Contents and Software Description

Hardware

The *CTrac* emulation hardware consists of the *CTrac* emulation printed circuit board, and an interface cable.

Software

The CTrac emulation software consists of the following files and directories:

Emulate

The emulation software

Builddisc

A tool used to create entire CD disc images for the CTrac emulation system. It combines track images, along with the proper subcode information, into one composite disc image. The final disc image is needed for the emulation software.

Buildtrack

A tool used to create ISO 9660 formatted tracks that are needed on the CD-ROM discs used in the CDTV. The output of this program is used as input for the BuildDisc program.

ISOUtl

A tool used to extract, update and list directories of files contained in ISO track images or from disc images that contain ISO tracks.

Libs

The Libs directory contains the requestor library that must be installed on the Amiga before running the emulation software.

QFS

The QFS directory contains further directories and files needed to run the Qwik File System on the Amiga. The Qwik File System was developed by CONSULTRON. The data rate and seek time requirements of the emulator exceed the abilities of the Amiga Fast File System. Therefore, the Qwik File System is provided and must be installed on the hard drive that will contain the disc images that will be used for emulation.

The Qwik File System is approximately 16 times faster than the Fast File System when performing directory operations and approximately 2 times faster when reading files greater than 36K. The Qwik File System must be installed on the hard drive that will contain the disc images to be emulated.

The Qwik File System is compatible with the AmigaDOS file requirements with the following exception:

16 character file/directory comments must be used instead of 79.

Install

A batch file that copies the emulation software and tools to the command directory of the Amiga boot drive.

The ISO DevPak diskette is also delivered with the Emulator. This diskette provides other tools necessary for the creation of the ISO 9660 image.

Hardware Requirements

Amiga 2500, Amiga 3000, or Amiga 3500, with one free expansion slot

The *CTrac* emulation software was designed to run on an Amiga 2500 or higher model, such as the Amiga 3000. If an Amiga 2500 will be used for development, the emulation software must be run in the 2500's 68030 mode. 3 Mbytes of RAM minimum are required.

Two or three hard disk drives, with one 600 Mbytes or larger

At least two hard disk drives should be used with the *CTrac* emulation software. One hard drive should utilize the Amiga Fast File System and contain all of the standard Amiga libraries and tools. The second hard drive, which should be the larger of the two, must have the Qwik File System installed. It will contain the disc images that will be emulated. If the image is too large, you may need a third drive.

CD-ROM discs may contain varying amounts of data up to a maximum of approximately 650 MB. Since the image of a CD-ROM disc is stored in a file on the hard drive of an Amiga, it follows that the hard drive used must be at least as large as the largest image that will be emulated. Moreover, it may be desirable to store the source data and track images that were used to build the disc image on the same hard drive. If this is done, it could more than double the amount of storage space required. Therefore, a 600 MB or larger drive is strongly recommended.

For example, assume your application requires 400 Mbytes of data and code. In this case you would probably need three hard disk drives, or a total capacity of 1200 MBytes.

- Drive 1, formatted under the Fast File System, contains your original source data and code. You develop and test your application on this drive.
- Drive 2 will receive the track image built by the BuildTrack utility. This image will be at least as large as your application code plus data (400 MB, in this case).
- Drive 3, which must be formatted under the Qwik File System, will receive the disc image built by the BuildDisc utility. Again, this image will be at least as large as your application code (400 MB).

One Drive May Be All You Need. You may combine both the track image and the disc image on one drive, if they will fit. If space allows, you can even build those images on your source drive, if you have formatted your source drive with the Qwik File System. Thus if your application contains only 30 Mbytes of code and data, you could develop it and test it under the emulator with a single 100 Mbyte drive. You may also use separate partitions of the same drive.

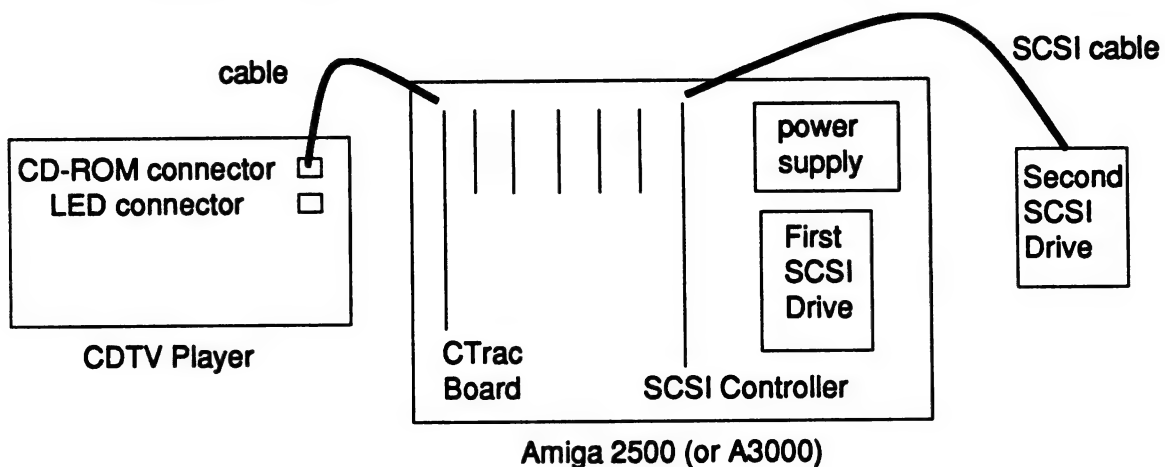
The hard drive containing the disc image must also be reasonably fast because the emulator requires at least a 175K bytes/second maintainable data rate. For example, we have tested the drive using the Quantum 40 and 100 Mbyte drives delivered with Amiga 3000s. Unfortunately, these drives are not fast enough to allow proper emulation. The emulator reports error messages indicating that it could not read data fast enough from those drives. We have successfully used Seagate 450 Mbyte drives (reference ST2502N 94241-502) to resolve these problems.

Installation of the Emulation System

Installation of the system is fairly quick, requiring approximately twenty minutes. First, the large SCSI disk must be formatted under the Qwik File System.

Next, the install program is executed to copy the tools necessary for image building and emulation to the Amiga hard disk drive.

Finally, the *CTrac* emulation board must be installed in the Amiga 2500 or 3000, and connected to the CDTV player. The figure below indicates the physical set-up of the emulation system.



Creating Disc Images and Using the Emulation Software

The following section describes the steps involved in preparing a disc image for the emulator, and using the emulator board and software to emulate that image.

For this discussion, we will assume that the application is in "mixed mode", that is, it contains a certain number of CD-DA audio tracks, as well as Amiga code and data. We will also assume the application has been written, and the data is ready. The application resides on a drive we shall name DH0:. A second hard disk, named Q0:, has been formatted under the Qwik File System to receive the image files.

Step 1

Use the iso utility to create a control file for *BuildTrack*.

The ISO utility is provided on the ISO DevPak diskette, available to licenced CDTV developers. It reads the directory structure of your AmigaDOS drive (DH0:) and creates a file which specifies which files to include in the ISO 9660 track, and the order of those files. The iso utility allows files to be sorted by size, alphabetically by name, or to remain in the same order as they were on the source disk. Iso creates an ASCII text file, which the developer can modify for further optimization.

Step 2

Use *BuildTrack* to create a track image.

The *BuildTrack* utility reads in the controlfile created by the iso utility. It verifies for ISO 9660 compatibility (maximum levels of nesting for subdirectories, filename conventions, etc.) It then generates an ISO 9660 track image on the target drive (Q0:)

Step 3

Use the *FixTM* utility

The *FixTM* utility, also on the ISO DevPak diskette, modifies the ISO image to allow for direct booting on CDTV.

Step 4

Create the *BuildDisc* controlfile.

Like the *BuildTrack* utility, *BuildDisc* is also directed by a control file. The control file, a standard text file, specifies which tracks will be included in the disc image, and in what order. It also allows other attributes of the disc to be specified, such as the length of the table of contents (TOC).

Step 5

Use *BuildDisc* to create the disc image.

BuildDisc reads the controlfile, and generates a disc image with the track images that were specified.

Step 6

Run the emulator using the disc image.

Now we can begin the actual emulation. The emulator software runs in the memory of the Amiga 2500 or 3000. It requests the name of the disc image file which should be emulated, and then begins to work.

Step 7

Reboot the CDTV.

Finally, we power on (or reset) the CDTV. The emulator software (along with the emulator board) will intercept all attempts by the CDTV to access the CD-ROM drive. They access the disc image on the SCSI drive instead. Furthermore, the board and software assure emulation of the data transfer rate and seek times of the CD-ROM mechanism.

The CDTV application can be viewed on a monitor attached to one of the CDTV video outputs. Meanwhile, the emulator software displays activity messages in its Activity Window. All accesses to the ROM are indicated, as well as all commands received by the emulator.

A Status window at the bottom of the screen displays the name of the image file being emulated, any emulator errors that have occurred during emulation, and the current state of the drive status bits (ready, audio, done, error, SpinUp, DiscIn, InfErr).

Optionally, time emulation can be suspended. If this is done, the emulator no longer emulates spin up, spin down, and seek delays. This permits testing applications at the fastest possible speed, but the emulation is no longer accurate.

Follow The Script. It is possible (and even advisable) to combine all these steps into a single script file. You can launch the script, let it run as long as needed, and return to test your emulation.

Emulator Limitations

Seek Times

The emulator approximates the seek times and other delays of the CD-ROM drive in the CDTV. Due to the variances in drives and discs, it is possible that a seek time on the emulator could be slightly different from that on the actual drive.

Tasks

In order to obtain optimal results, it is not advised to run other tasks on the Amiga 2500 or 3000 simultaneously with the emulator software. The emulator needs a large amount of CPU time, and its priority is set to 100. Furthermore, contention may develop between two tasks trying to use the SCSI bus simultaneously.

Disk space and speed requirements

As discussed above, the *CTrac* system requires large amounts of SCSI disk space—three times the size of your application. Furthermore, the disk must be a fast one, if you want to accurately emulate the data transfer rate used by the CD ROM mechanism of the CDTV player.

Speeding Up Your Titles

The Importance Of Speed

The demands of multimedia titles are different from the demands of traditional computer programs. Computer users are familiar with the idea of waiting for data to be loaded, particularly when working with floppy disks as the storage medium, but users of multimedia applications approach them in the same way that they watch a videotape of a film. Long delays with no apparent activity will soon make the users hostile to the title. They are not interested in *why* it takes ten seconds to load an animation—they want the show to go on.

Some thought must therefore be given to convincing users that the title is doing something sensible, or they will begin to stab at the buttons on their infrared controller or keyboard.

Usually, and understandably, these long waiting periods result from programmers applying quite proper traditional wisdom to a medium whose physics and demands are a little different.

Subjective Speed

A common delay on a CDTV title occurs when the user presses a button to go on to the next picture or screen in a series. Typically the title will wait until a button is pressed, and then begin loading the data. This will take *at least* one second per 150 KBytes of data, plus any seek times involved in the laser moving to the position of the data in the CD-ROM track.

If, however, that picture is loaded into a second screen buffer while the user is looking at the previous picture, it can be made to appear on the screen instantly when the user presses the button. The load time for the data has not changed, but the user's subjective impression is that the system is very responsive.

This simple technique can change a sluggish title into a snappy title with no low-level optimization of data transfer rates, but it does require planning at the design stage to make the double buffering possible. Most approaches to improving performance depend on good design at the planning stage and cannot simply be *bolted on* afterwards.

The Startup Sequence

The first impression a user has of a title is what happens when the disc is inserted. The following sequence of events takes place:

1. The system detects a new CD and resets.
2. The system examines the disc to determine if it is a CD-ROM disc or a CD-DA disc.
3. If it is CD-DA, the CD player screen is launched from ROM.

4. If it is a CD-ROM disc or mixed-mode disc (and there is no floppy disk drive with bootable disk attached), the system tries to boot the disc.
5. If there is an ISO-9660 image in track one, this is examined. If it is a CDTV CD-ROM and it has the correct copyright data in place it is booted. If the copyright is not correct, the system turns the screen red.
6. At this stage the startup sequence in the S directory of track one is executed.

It is clear that this process takes several seconds to perform before the startup-sequence itself is activated. So that the title itself can begin quickly, some thought needs to be given to the startup-sequence.

It is not normally necessary on CDTV to use an entire Workbench startup-sequence. Assuming no *Assign* statements are needed the minimum startup-sequence may be:

```
rmtm      ; remove trade-mark
MyTitle   ; executable for my title
```

A more typical Workbench 1.3 startup will need at least the following components (in no particular order):

```
rmtm      ; remove trade-mark
setpatch >NIL: r      ; patch layers etc
FF >NIL: -0           ; speed up fonts
Assign MyVol: CD0:    ; assign logical volume that was hard disk
                  ; to CD
bookit bv            ; blank screen, centered Views
MyTitle             ; run the title!
```

With these actions in the order above, the *rmtm* program will delay a few seconds and fade the copyright notice. The screen will (if the preferences set in *devs:system-configuration* are black screen *and they most likely should*), then be black. The CDFS then finds and launches *setpatch*, *FF*, *Assign*, *bookit*, and finally *MyTitle*. This involves perhaps ten seconds of black screen before *MyTitle* has a chance to run its title music and display its title screen.

The following order of events reduces the black screen considerably:

```
setpatch >NIL: r      ; screen shows trademark
bookit bv            ; screen shows trademark
FF >NIL: -0           ; screen shows trademark
Assign MyVol: CD0:    ; screen shows trademark
rmtm                 ; trademark fades
MyTitle              ; black screen until titles appear
```

Your title may require a substantial load time, particularly if it involves invoking an authoring system which must set itself up before it begins to run your series of commands.

The black screen can be eliminated almost entirely by using the *keeper* program. *Keeper* will display an IFF picture of your choice (probably your logo) until another View is loaded, or it is signaled from within your program, or it is launched a second time with the parameter *QUIT*. The best position for the use of *keeper* depends on how long your title needs for its initialization, and how tolerant it is to another View occupying display RAM. It *must* follow the *rmtm* command.

Games Unkeep The Keeper. There is an implicit *LoadView* whenever a program opens a screen and displays it. The only exception will be some games programs that address

the display hardware directly. Such programs will remove the *keeper* picture before they start.

A title that requires as much Chip memory as it can sensibly expect to have will load with the following example startup-sequence:

```
setpatch >NIL:
bookit bv
FF >NIL: -0
rmtm
keeper CD0:logo/MyLogo.ILBM
Assign MyVol: CD0:
MyTitle
```

As soon as *MyTitle* opens a display, the *keeper* task will terminate, close its *View*, and hand back to *MyTitle*. To get the maximum screen memory it can, *MyTitle* will, during its initialization, include the following code, which assumes a procedure called *GetDisplayRam()*.

```
/* various initializations */
task = FindTask("PicKeeper");
/* assuming task is found */
Signal(task, SIGBREAKF_CTRL_C);
GetDisplayRam();
/* load and display titles etc */
```

If an authoring environment is used it will be safer to use the following outline startup-sequence.

```
setpatch >NIL: r
bookit bv
rmtm
keeper CD0:logo/MyLogo.ILBM
FF >NIL: -0
Assign MyVol: CD0:
keeper QUIT
MyTitle
```

The black screen time here will be precisely the time needed for the title to boot and open its first screen. If you are certain that your title leaves enough memory free for the *keeper* *View* and can still start properly, simply omit the second call to *keeper*.

There is no simple guide for startup-sequences except the need to organize events in such a way that all initializations can be done without the user wondering if the system is dead.

Editing the startup-sequence is usually the last stage before pre-mastering a CDTV title. It has some unpleasant possibilities to it. For example, a tiny slip in an *Assign* statement can produce an expensive and useless disc. It should therefore be thought about in advance of the pre-mastering, use the minimum set of commands possible, and interleave these with the *rmtm* and *keeper* programs to produce the least user irritation. Some things will have to be guesswork. When running a title on hard disk, the *bookit* program cannot be used. When a CDTV is booted from a floppy drive, the trademark sign does not appear. It is a good idea to design the startup-sequence before making the first Check Disc, even if this is intended to be used with a floppy or SCSI drive for further development. That way, you can be reasonably sure that the loads occur in the right order and with sensible timings.

The Preferences settings in *devs:system-configuration* are just as important as using *bookit*. It is always safest to have completely null Prefs fields. For example, *bookit* will remove the pointer during its life, but it will reappear as soon as the main program starts. Blanking the pointer should not, therefore, be left to *bookit*.

Distracting The User During Loading

There will be moments when data must be sought and loaded from the CD. The users may reasonably expect that they will get some clues about what is happening. Expecting them to watch the front panel light is not an adequate feedback. The traditional computer world solutions to this problem include:

- Nothing.
- “Busy” icon.
- “Loading—please wait”.
- Writing dots to the display periodically.
- Graphic bar that fills as the load progresses.

Most CDTV titles will be multilingual and avoid any language specific screen prompts.

“Busy” signals and sound jingles are the easiest solution. Fancy animations to cover loads may take longer to load and activate than the load time for the data they are masking. Jingles that occur frequently can be irritating. It must be obvious to the most naive users that they are waiting for something to arrive.

A good “busy” signal screen will have some animation, if only color-cycling, and indicate that a request is being satisfied. Shopping may provide a good analogy. You order a CDTV from a shop assistant who vanishes and *stays* vanished; you wonder if you have been forgotten and eventually, you leave the store and buy the CDTV somewhere else. In a good store, the assistant will telephone the order to the stock room and then distract you by showing you a whole range of CDTV titles you may wish to buy. At least the assistant will say “This will take about five minutes. Why don’t you look around and I’ll tell you when your CDTV is ready for collection.”

CDTV does not have a standard “busy” iconography like the “Zzzzzz” on Amiga Workbench 1.3 or the hourglass on Windows™. Good equivalents on CDTV are spinning CDs, graphics of movement along rows of books on a bookshelf, elevators rising through several floors of a building. Since they will be used often, they will be stored in RAM as *sbox* items and unpacked with the *debox.library* functions when needed. In complex titles they will be context dependent, and show that a map is being found, or a novel, or a musical composition.

The target should always be to minimize load times and inform users that their choice is understood and is in progress.

Directory Structure And Disc Geometry

CDTV uses an international standard for directory and data arrangement known as ISO-9660. Because the CDFS understands this system, it is *not necessary* for programmers to worry about it; very few titles will address the ISO-9660 structures directly, but will rely upon the CDFS to handle this.

For optimization of CDTV performance it is valuable to understand the basics of ISO-9660. The ISO-9660 filing system is very simple. It is designed for a *read-only* medium and need not involve the complexity required for file writing and update. This simplicity may be harnessed to achieve the best possible performance from the medium.

Normally all files will consist of contiguous blocks of data. Each block is a logical sector of 2,048 bytes. (*Other sector sizes are allowed, but the current buildtrack software uses 2048 byte sectors*) Files always start on sector boundaries, so that a file's starting position from the beginning of the ROM track will always be modulo 2048=0. *The ISO-9660 standard does allow for file interleaving, but this will be a very unusual state on a CDTV title.*

The root directory contains pointers to files at a logical block. A file may contain data or a directory. The system is hierarchical, and permits directory nesting to a depth of eight. (This restriction does not apply to the CDFS, which has extensions to allow many levels of directory.) In addition to the root directory block there are "path tables". Path tables allow quick access to directories, and are usually held in memory by the CDFS. It follows that the most efficient file access under CDFS happens when the path table can be used. This has a consequence that may appear strange.

Many Small Directories Beat One Large Directory. It is more efficient to use many directories containing a small number of files than one directory containing many files.

Data should be organized to ensure that the laser has to move the smallest possible amount between seeks.

The system performs best on single files, so that a complex file which contains all data (sound, pictures, notes etc.) may be more efficient than a group of files containing separate component elements.

In practice, only file transfers that require the highest level of optimization (e.g., CDXL files) will be forced to cluster their data inside one file. Problems arise when reading from multiple files when file A is at the hub of the disc and file B is at the periphery, because the laser may take up to 0.8 seconds to make its physical move between the inner and outer edge of 600 MByte disc.

The performance of a title on hard disk is not a reliable guide to its performance on a CD-ROM. The organization of data under the Amiga Fast Filing System (FFS) on a hard disk is radically different from the organization of data on a CDTV CD-ROM. Generally the hard disk will be much faster, but this may not be the case with heavily fragmented files on large drives with slow seek times. One of the advantages of the ISO-9660/CDFS system is its regularity and predictability. There is *no* fragmentation.

A good question, then, is *How do I lay out my directories for best performance?*

The *iso* software on the Iso_Devpac disk creates a text file called "controlfile". This text file is used by the *buildtrack* program to create the ISO-9660 directory structure, path tables, and data blocks exactly as they will be arranged on the CD-ROM. The controlfile can be edited with a text editor. The rules are simple: directories can be rearranged, file order can be changed, but files may *not* be moved out of their directories. The *iso* program will pick up the file order from the AmigaDOS file system on the hard disk. Since it is not possible to reorder directories on working hard disks, the order may be arbitrary, or alphabetically sorted, or sorted by size (the default for files in a directory).

If we assume that directory Alpha contains data closely coupled to directory Theta, it is desirable on the CD-ROM to have the data files in these directories as close together as possible. Note the distinction here—the directories themselves are merely system pointers to data segments of one or more sectors on the CD-ROM. However, *buildtrack* will allocate data segments to the files in the order it is presented with the files. By keeping the directory's file count low and positioning closely-coupled directories together in the controlfile you can ensure that the data will be reasonably

close on the CD-ROM, and so eliminate unwanted laser moves between file accesses. You can also change the order of files within a directory to minimize head moves between file accesses. The relatively slow seek time of a CD dictates that getting into a "disc thrashing" situation is something to avoid at all costs.

Compared to the working methods used on hard disks, this has some evident advantages, i.e., almost everything is under your control. The cost is a little extra effort after the data gathering and programming are finished. Attention shifts to disc layout and optimization, and significant time should be allowed for this in project planning.

A number of strategies exist for ensuring that the data sets are as close as possible.

Buildtrack

The "buildtrack" software can make an ISO-9660 image into an AmigaDOS file. This file can then be examined with the *isodir* program. *Isodir* provides a full listing of each directory and file's position in the CD image. Where the program picks up a group of files in close association, by merely checking the start sector for those files it is possible to calculate approximately how long it will take to seek to each file before transfer of data can begin. We may discover, for example, that a gap of 200 MBytes exists between file A and file B because there are several large directories in between. This gap will cause a significant delay, and if file C lies close to file A, then the traverse will happen a second time.

Naming Conventions

A second, less labor intensive technique is to apply naming conventions for directories and files to ensure that an alphabetic sort will place the components in the best order.

Match Data Organization To Data Presentation.

An optimum CD geometry exists for any title, but this may be impossible to determine *a priori* if thousands of branching decisions are made by the user. Normally, it will be possible to assign priorities to data sets. For example, if a title is making a major change of focus, as for example, when an Atlas program changes country or continent, the user may tolerate some waiting; it is intuitive to think of going from one continent to another as involving delay. If that title takes the same amount of time to go between adjacent streets the user will probably be irritated. This is a case where keeping all mapping information in one directory, all text in another, all sound in a third etc., is not helpful. It is better to break the directory structure down by continent, country, and town, ensuring that the big laser moves happen where the granularity of the data is largest. This requires no significant adjustments of the directory order on the CD. When fine tuning the title, it may simply be desirable to change some file ordering, but even that is less likely.

Speed Up by Duplicating

The perceived speed of a title may also benefit greatly if frequently needed menus, pictures, or sound data files are duplicated in such a way that the current directory contains a copy. If all the menus are neatly tucked away in a directory called "menus", this is a traditional, sensible and logical way of ordering data, but it may cause unwanted tracking backwards and forwards.

Keep Directories Small

For highest efficiency in the CDFS buffers directories, the number of files in one directory should be restricted to forty.

Using CDXL To Load Data

Let's say that at a certain point in a title's progress on CDTV, it requires the following data to be loaded into memory:

- 96,000 bytes of sound data.
- 2 pictures—64,032 bytes and 128,016 bytes.
- 938 bytes of text data.

This will require four calls to the AmigaDOS `Open()`, `Read()` and `Close()` functions, together with error checking. Despite the efficiency of the CDFS, there will be a noticeable break between each load. If the component data objects are widely separated on the disc, there may also be some unwanted seeks involved.

If, at the simplest level, these files were to be concatenated with the *Join* command, the data would be contiguous and would require only one `Open()` and `Close()` for access. However, it would still require four `Read()` calls to the various buffer positions.

The CDXL commands to *cdtv.device* allow a high speed DMA transfer of this data. (For a full explanation of CDXL, refer to the "CDXL Overview" article in the CDTV Specifics subsection.) The price for this gain in transfer efficiency is extra effort in design. It will be necessary to write "software tools" to arrange the mixed data into a file and include the necessary information about the size, order and nature of the data chunks within this structure.

The order in which the CDXL transfer list is constructed depends on what needs first to be displayed. For example, assuming that the 640 x 200 x 4 picture is needed on the screen as quickly as possible, it is placed first in the transfer list. The `DoneCode` callback item in the transfer list item for that picture will not display the picture itself, but will signal the main process that the data has arrived. The CDXL transfer will continue with no interruption:

```
void PicReady()
{
    Signal(MyTaskPtr, MySigBit);
}
```

Having established a way of finding out when the picture is ready, the main process can initialize the CDXL transfer of the data and do other things until the picture actually arrives:

```
SendIO(MyIoRequest);          /* send CD_READXL command */

/* additional actions as required */
sigs=Wait(1<<MySigBit);      /* picture ready when Wait completes */

LoadRGB4(&MyViewPort, &BufferColorTable, NumColors);
LoadView(&MyView);
/* picture now on screen and transfer still running */
```

This is the fastest way to get complex data from the CD-ROM into memory. Given that multimedia applications often have associated sound, pictures, and other material, it is well worth the extra effort (which is not great) in setting up the linked transfer list. The major effort involved is in getting the associated data into one file, or if not into one file, into a series of contiguous sectors on the CD-ROM. This technique absolutely depends on the data elements being sequential.

A more complex extension of this approach to data management is to slice up sound and picture files and load a series of pictures transparently while a continuous stream of sound is being spooled

from the CD. This demands a very tight control of data sizes, sound playback frequency and other timing considerations.

As a general principle, optimization of CDTV titles frequently depends on shifting the burden of computation from run-time to the development stage.

Using Burst Mode

Burst mode is not a special function that you can switch in or out, and the term can give rise to confusion. The CD mechanism and circuitry buffers sector transfers internally before sending the data in a "burst" onto the data bus. This happens transparently. The consequence of it is that large continuous transfers of data are more efficient than many short transfers. A second consequence is that data transfer is not a linear time function.

Avoiding ILBMs

The IFF ILBM file format for pictures has been very successful in promoting easy exchange of pictures between application programs. The first line from each of the picture's bitplanes is run length encoded, and then the second line and so on until the whole image has been converted. The advantage of this scheme is that if a picture is decoded from a slow storage medium, each successive line can at least appear with all its color and information intact.

Unfortunately on an unexpanded CDTV, the ILBM format requires the 68000 to parse the IFF file, unpack the rows, reorder the color map and do various housekeeping tasks. The 68000 is running in Chip memory, and is losing cycles to the custom chips and DMA. The result is a distinct delay before the picture appears.

An older IFF picture format, the ACBM (Amiga Contiguous BitMap) provides a partial solution. In an ACBM, the BODY chunk is replaced by an ABIT chunk which contains the data for each bitplane in flat format. An Amiga Basic program called *SaveACMB* will convert ILBMs to ACBMs. Developers using authoring systems should check whether their authoring system will read ACBMs. If it will, substantial speed increases can be gained by using this format.

Developers who are coding titles themselves may want to consider whether they should bother with the IFF format at all once the graphics work has been completed for pictures other than ANIMs. If the title uses an arbitrary mixture of picture formats, it will be difficult to dispense with the header information, but the more commonality that is imposed on all the data types used, the greater the potential optimization available.

Generally, Intuition will not be used in CDTV titles because it is not designed for viewing and operating from a distance of ten feet on a television screen. Programmers are more likely to create their own custom **Views**, **ViewPorts** and **RastPorts** and render directly to these. From here it is a small step to an arrangement of bitplane and color information that allows a single DMA transfer of the data with no interruptions and no parsing.

Contiguous bitplane data for picture	Color Information	8 byte spare RAM (extra bytes to avoid DMAC problems)
---	-------------------	--

If the file format adopted conforms exactly to the layout shown above, the fastest possible load time is obtained. This scheme relies on programmers taking control of their own graphics structures. If memory requirements are tight, it is desirable in any case to manage an internal heap for picture and sound buffers, with a variety of Views managing different display ratios.

For example, if 327,744 bytes of Chip memory are allocated at start up, the space will allow for two 640 x 512 eight color picture buffers to be available at any time, or five 320 x 256 HAM pictures. The load-and-display routines can be organized to give double-buffered displays over a wide range of picture formats with very little processor use and no possibility of out-of-memory problems. Unless a CDXL scheme is used for loading pictures, the normal scheme becomes reversed.

Normal

- Load Header.
- Examine Header.
- Set Up Screens And Windows.
- Move BitPlane Data Into Buffer.
- Apply Color Mapping.

Optimized

- Select Appropriate Buffer.
- Load All Data.
- Use ViewModes To Determine RastPort & ViewPort.
- Load Colors.
- LoadView().

The price for this is that the picture format is no longer compatible with paint programs and changes may be difficult to make. So when designing software tools to take standard IFF format files and transform them into optimized objects, it is worth the small extra effort required to make this a two-way process.

Ideally the graphics engine for the title will be capable of loading all formats, and a very late stage in development will "batch process" the picture files into the optimized layout.

Asynchronous Reads And Multiple Reads

Where possible, the multitasking power of the Amiga/CDTV system should be used to reduce *perceived* loading times. One obvious example of this is asynchronous loading of data. While something wonderful is happening on the screen, the next picture or sound is invisibly moving into memory ready for rapid display. This can be achieved at three main levels:

- Process
- Task
- Device

The advantage of a process over a task is that a process can communicate with DOS and a task cannot. Processes may be spawned from inside the main program, and talked to with signals, semaphores, or messages. Alternatively, a process may be a separate program that is *Run* from the startup-sequence, has a named Port, and hence may be sent messages from several multitasking programs that constitute the title. This approach will be successful for relatively simple cases such as audio management, but may become very complex for control of pictures.

Tasks are simpler (though not much simpler) to launch from within a program. Because they are “locked out” from DOS, however, they lose the simplicity of being able to find and load a file. Tasks can use the system devices—*audio.device*, et al.

The power of CDTV resides on the strength of the *cdtv.device*. The high-level facilities of the DOS interface rely on low-level commands to *cdtv.device*. The *cdtv.device* is not a filing system, and consequently does not respond to file names or other file attributes. It requires commands to transfer data at a logical rather than symbolic level.

The decision that programmers make about when to use a process, a task or a device call depends on the degree of direct control they require, or the degree of binding between the main program and the loader code. The use of processes and tasks provides generality, but requires very careful control of task/process priority. Particularly under Kickstart 1.3, it is possible to create circumstances where a separately launched task will *never* run. There may be circumstances at run-time when different priorities apply. “Race conditions” can arise where the order of completion of operations can be very complex under different loadings.

Using processes and tasks for asynchronous data handling is appropriate in a situation where, for example, an authoring system allows external calls via ARexx but does not provide the facility to handle large stereo sound files. It is unlikely to be the ideal solution in a complex multimedia program which is seeking the highest possible control, because the order of operations has been handed over by the programmer to the Exec task switching mechanism.

Asynchronous data loading with the *cdtv.device* is a simple matter provided that the location of the file on the CD-ROM data track and its size, is known or can be determined. These two items of information are available through the CDFS.

The first sector of the file is obtained by calling the DOS `Lock()` function with a pointer to the name and the access mode required, which is certain to be `ACCESS_READ` on a CD! Assuming the file exists, `Lock()` returns a BCPL pointer to a lock. After conversion to 680x0 pointer, the `Lock` is in fact pointing to a `FileLock` structure in memory. The second field of the structure is `fl_Key`. This is a `LONG` (32 bits) giving the sector where the file starts. No further information is needed to locate the data because the file will not be fragmented.

The size of the file can be obtained by passing the original BPTR to the DOS function `Examine()`, which also requires a pointer to a `FileInfoBlock`. The `FileInfoBlock` structure field `fib_Size` contains the file size in bytes.

A Global Structure Makes A World Of Difference. Obtaining the location and size of a file by the method described above cannot be performed from a task, because it requires DOS access. If your asynchronous loader is a sub-task, it should therefore be allowed to access a global structure in which the main process places the location and size before signaling the task, or be sent a `Message` which includes the information.

If this file will be needed frequently, it may be useful to store the position and size values, and in many instances where programmer-defined data structures of fixed size are involved there may be no need to call `Examine()` to determine the size.

With the information to hand, one command to *cdtv.device* will load the file, or part of the file.

Assuming an `IOStdReq` structure has been allocated, initialized, and opened successfully on *cdtv.device*, the following example performs a fast asynchronous load:

```
#define SECTOR_SIZE 2048;

MyIOReq->Offset = StartSector*SECTOR_SIZE;
MyIOReq->Length = FileLength+8;
MyIOReq->Data = &MyBuffer;
MyIOReq->Command = CD_READ;
SendIO(&MyIOReq);
```

Notice that `SECTOR_SIZE` may (rarely) be different on different CD-ROMs, but will currently be 2,048 bytes for CDTV discs pre-mastered with existing utilities. The extra eight bytes added to the Read length is vital for direct transfers, and in this instance `MyBuffer` needs to be eight bytes longer than its significant size. This is because the DMAC system may very occasionally (1% chance) corrupt the last eight bytes of data transferred.

While the `SendIO()` is pending, other operations, including preparing and sending further requests to the *cdtv.device*, may continue—monitoring user I/O, screen painting or playing memory resident audio samples. Periodic calls to `CheckIO()` will report the status of this I/O request, and finally `WaitIO()` will put your program to sleep until the operation is complete.

Given a handful of `IOStdReq` structures, a series of such commands may be queued to the device. However, where the number of queued items or the complexity of the data becomes significant, it may well be easier to set up a CDXL transfer list. But for most load operations, it will be significantly faster to use the `CD_READ` command than to rely on CDFS which has to maintain a broad accessibility.

Unfortunately, *cdtv.device* I/O will not work on a hard disk. It requires the presence of a CD-ROM, and so usually will be brought into operation once the data are finalized and available on a Check Disc CD-ROM.

Optimization Summary

There is remarkably little that can be done to optimize titles created with authoring systems that do not include the relevant low-level facilities, except scrupulous attention to directory layout and disc geometry.

For those not using authoring systems, the following summarizes what has been discussed.

- All titles, using whatever optimization, are vulnerable to poor layout.
- Custom file formats and internal memory organization can create a system in which data loading can be optimized.
- Asynchronous loading of data which is known to be required next can eliminate many delays.
- The *cdtv.device* provides all the facilities needed to ensure that the flow of data into memory is efficient in the foreground and possible in the background.
- Where possible, computation should be performed “off-line” during development.
- Careful thought given to data formats can simplify programs and radically improve performance. This is a data-driven system.

Optimal Disc Layout

A primary key to improving the performance of a CD-ROM-based application is to eliminate as many seeks as possible. While this can be done by economizing the title's disc accesses, many further gains may be realized by thoughtful layout of the files on disc. With some knowledge of the ISO filesystem structure and taking advantage of the CDTV Device File System's (CDFS) caching, significant speed gains can be achieved, improving title performance.

ISO Structure

The structure of the ISO-9660 filesystem is covered in more detail in the "CDTV File System" chapter. However, only some basic facts are needed to understand the techniques that follow.

ISO-9660 requires that files occupy contiguous sectors on the disc, permitting the entire contents of the file to be scooped up in a single operation. Because of this, there are no "side-sectors" as there are with AmigaDOS. Thus, there is no secondary seeking to collect block lists. The filesystem can go straight from the directory entry to the file data.

ISO directories can also be thought of as a special form of file. Unlike AmigaDOS, which reserves an entire disk block for each file descriptor, ISO packs together as many descriptors as will fit in a single block. Directories that consume more than one block are stored, like files, in contiguous blocks. In this way, an entire directory can be read all at once.

In order for a file to be found, the directory that contains it must be scanned. Directories themselves, however, can be found by scanning the path table, which is loaded by CDFS upon disc insertion.

Cache Operation

CDFS has a rudimentary block caching system. Two sets of caches are maintained: one for data blocks, and one for directory blocks. The sizes of both caches can be set through the Boot Options (covered in the "CDTV File System" chapter).

The data block cache is a simple read-ahead cache. If a requested block is not in the cache, the entire cache is dumped and refilled with the requested block and the N blocks following it. Thus, the read-ahead cache's greatest value is realized by doing sequential reads.

The directory block cache is a bit more intelligent. When a directory is loaded, its blocks remain in the cache until the cache is full. If filled when a free cache entry is required, the least recently used entry is dumped and recycled. When a new directory block needs to be read, it is requested through the above-described data cache. Thus, a directory scan can cause the data cache to be dumped. Note however that, if directories are grouped together, the data cache gets dumped only once. Any further directory block requests that immediately follow are satisfied from the data cache.

The cache operation is described here only to give you the most general of ideas as to how organize your discs. The actual internal workings of CDFS are private. Commodore reserves the right to alter the caching mechanism in the future to improve performance.

Analyzing Your Title

Before you can knowledgeably lay out your disc, you must first know which files and directories your title accesses, and the order in which it does so. Some of these file accesses are non-obvious. An example would be the DOS `Execute()` function, which first loads `C:Run` before loading the specified program.

One tool that can help is *PickPacket*, a public domain program that wedges into any DOS device and prints a report of all DOS operations requested of the device. *PickPacket*'s output can be used to more accurately determine your title's sequence of reads and file accesses. You can also use the Commodore-supplied tool *OptCD*, which uses the built-in statistics-gathering features of CDFS V26.

Goals

Finally, before you start laying out files, you should decide what you're trying to accomplish. Obviously, you're trying to optimize disc accesses, but optimizing for one case may leave other cases in very poor shape.

While collecting the raw data, use the title yourself (or better, have a disinterested party do it) and determine which parts seem "slow". In particular, ask these questions:

- Does the title start slowly (more than 5 seconds)?
- Does moving from one area to another seem to take a long time?
- If it has a search facility, does it take a long time to find something?
- Is there any part of the title that seems to take longer than it should?

If you feel the title is slow in a specific area, you may wish to pay particular attention to the reported disc activity when the title is operating in the area of interest.

Disc Layout

Once you have an idea of where your disc accesses are occurring, and which ones you wish to optimize, you can start laying out files. Though you can't specify a particular block number when creating an ISO image, the mastering software will let you specify the order in which files and directories are laid out, giving you a high degree of control over file grouping.

While no single approach will yield the ultimate optimized disc, some general guidelines will yield good results.

The startup-sequence can be the biggest offender in your boot times. Group all the commands that appear in the startup-sequence together on the disc in the order in which they appear in the startup-sequence, and put the startup-sequence itself immediately before all of them. This will cause the commands to be pre-loaded into the read-ahead cache when the startup-sequence is read. Keeping your startup-sequence short also helps a great deal.

Group small files together (8K or less), and group them in the order in which they're accessed. In this way, when the first such file is opened, the rest will be pre-loaded by the read-ahead cache.

Also, consider recoding your application such that all the small files are accessed at once, rather than interspersed with accesses to large files. You may even wish to consider caching all such files in RAM, thus requiring them to be read but once.

Open all disk-based shared libraries at once, and group them on the disc together in the order in which they're opened.

Don't be afraid to have large directories on an ISO disc. Recall that ISO directory entries are packed together; thus, more than one entry can be stored in a disc block, and these blocks get cached by filesystem. A large directory, if accessed frequently, will remain cached. Opening files happens more quickly from a cached directory.

You may care to experiment with leaving system files (libraries, devices, fonts, etc.) in the root directory. At boot time, if DOS can't find, for example, a directory named "libs" in the root of the boot volume, it will assign LIBS: to the root, and all libraries will be searched for there. By doing this, you can keep the system from consuming directory cache entries for "libs," "devs," "fonts," "l," and "c." Note that there is a tradeoff here between more efficient cache usage and a really cluttered root directory. You'll have to weigh both sides and decide accordingly.

The Startup-Sequence *must* be located in a directory called "s", or the system won't find it.

If your title will have Workbench icons on it, we strongly recommend you group all the .info files together, and lay them out in alphanumeric order. ISO mandates that directory entries be sorted in ascending ASCII order, which means the DOS function ExNext() will encounter them in that sequence. Therefore, when Workbench starts searching a directory for icons, it will see them in alphanumeric order. If they are laid out in this way on the disc, all the icons will be loaded in a single contiguous read, and they will appear on screen *very* quickly.

If you have files which are seldom accessed, you might place them in a separate directory, and locate those files and the directory away from the more-frequently accessed files and directories. This way, you won't have a "speed bump" which needs to be skipped in the middle of your critical files. Keep in mind, however, that doing this can make these files harder to access, and could make operations involving these files slower than they were before.

If your title uses CDXL, you might also consider moving all the CDXL files in a separate directory and isolating them, too. Since CDXL doesn't go through the filesystem, it's to your advantage to have CDXL files located away from "normal" files.

You might even consider dispensing with the filesystem entirely and using CDXL for all your data-loading needs. It's extremely flexible and blindingly quick. Search your title for file operations that could be done using CDXL instead.

Disc Access

While disc layout is important, so also is the manner in which your title accesses the disc.

Make your read lengths as long as possible. If your application reads data in very tiny chunks (less than 1K or so each), the most efficient use of the system is not realized. The filesystem and the CD-ROM prefer to deliver the largest number of bytes possible. If your application requires piecemeal reads, consider incorporating buffered I/O routines. Most C compiler packages include them with their link libraries. Set your buffer sizes to 2K or more.

Try to minimize "file hopping" (switching frequently between two or more files) as it is seek-intensive. Try to process individual files in their entirety.



Pre-Mastering and Mastering for CDTV

Introduction

Preparation of titles for the CDTV player presents new challenges to the developer. Combining graphics, animation, text, digitized IFF sound samples and CD audio (CD-DA) samples in the same application provides endless opportunities for creation. Multimedia applications for CD-ROMs also require new production methods. This document will describe the various steps and techniques necessary for taking an application which runs on an Amiga SCSI hard disk and transferring it to a CD, eventually adding CD-DA sound along the way, and having that CD replicated.

Finally, we will discuss how the C-Track Emulator can simplify and expedite this process.

We will assume that the application runs correctly on an Amiga 2000 under Workbench 1.3. The code, data, and IFF sound files are on an AmigaDOS SCSI hard disk. The CD-DA sound tracks are on a DAT tape or, alternatively, on a 3/4" U-matic tape.

Overview of the Pre-Mastering and Mastering Cycle

Some developers may prefer to avoid the pre-mastering job. All approved CDTV pre-master centers will accept your AmigaDOS SCSI hard disk, a set of Amiga DOS floppies, or a set of floppies made with a backup program. Many will also accept AmigaDOS files on QIC-150 or other tape formats. There is a small charge for that service, and there will be a short turnaround time for the ISO formatting. The advantage to the developer is that the a pre-mastering center will have a fair amount of expertise in creating an ISO-9660 image and can create the image efficiently.

We can divide the process into a number of steps. Here follows a brief description of the steps. Each step will be explained in detail.

Preparing the Tools

Before starting, you should verify you have all the hardware necessary to complete the job as well as the appropriate software tools.

ISO Control File Builder

The ISO tool scans the directory of your AmigaDOS drive and builds an ASCII control file with the disc structure and file hierarchy. You may wish to modify and optimize this control file manually.

Creating an ISO Image

The Buildtrack tool reads a control file and creates a binary ISO image file. This file can be stored as a file on an AmigaDOS disk. Alternately, you can use the same Buildtrack tool to create a block-for-block ISO image on another SCSI hard disk. The *entire* SCSI disk will be used—you cannot create a block-for-block image on a second partition of an AmigaDOS drive.

Creating a Test Disc

The ISO image file must now be transferred to a pre-mastering system, such as a Meridian CDPro. The SCSI disk connects directly to the Meridian. The Meridian then prepares a test disc, using a write-once CD-ROM drive such as a Yamaha PDS. The Meridian can also accept DAT tapes or 3/4" U-matic tape for CD-DA.

Testing

The Pre-Mastering Center will return a test disc to you. You should verify this disc's operation in a CDTV through vigorous testing.

Mastering and Replication

Satisfied with your testing, you tell the Pre-Mastering Center to prepare a pre-master tape. This tape, either 9-track or 8 mm, containing a logically formatted ISO-9660 image, is sent to the replicator of your choice. The replicator creates a master stamper that is subsequently used to press the CDs. Replicators also provide packaging services and ship your CDs, in jewel cases or long boxes, to you.

Description of the Production Cycle

Preparing the Tools

Hardware

The basic platform necessary to the developer for production is fairly simple: an Amiga with at least 3 Mbytes of RAM and one or (preferably) two SCSI hard disks.

Amiga: You can use either an Amiga 2000 (or 2500) or an Amiga 3000. Remember that CDTV currently contains Workbench 1.3 in ROM, and your application must not contain any WB 2.0-specific code. If you have an A2000, make sure you use an A2091 controller. The A2090 and A2090A controllers may have difficulty in controlling more than one SCSI drive.

SCSI hard disk: We have verified and recommend using the following makes of high-capacity SCSI drives:

Seagate ST4766N	600 Mbyte formatted capacity
Seagate ST2502N	410 Mbyte formatted capacity

Preparing the SCSI Drives

Up to 6 SCSI devices may be daisy-chained behind one A2091 controller. For the physical connection, follow the instructions in the 2091 User Manual. Be sure to set the device number on each drive, avoiding two drives with identical numbers.

Use the HD tool box to partition your drives, and verify that they are correctly recognized.

Bulldtrack

Bulldtrack lets you create an ISO image file on two different devices. You can create a block-for-block image on a SCSI hard disk or you can store the ISO image file on your AmigaDOS hard disk as an AmigaDOS file.

The block-for-block method

This will usually be chosen by developers who do not have direct access to CD mastering equipment. Once the block-for-block image is created on a hard disk, you simply send the hard disk to a pre-mastering center. They will read the data onto their mastering equipment and prepare a test disc.

If you choose this method, the disk must be connected as SCSI device 5.

The AmigaDOS file method

This allows you to store your ISO image as an AmigaDOS file on any AmigaDOS device. This method may be preferred in the following situations:

- if you are in a networking environment, with the mastering equipment connected into the net. You can then transfer the image file to the mastering equipment via the network.
- if you are physically near mastering equipment, you can transfer the image file to the mastering equipment via serial or parallel lines.
- if you want to keep more than one ISO image file in your system, for testing or control purposes.

Software tools

The ISO Dev Pak diskette contains all the necessary tools to create an ISO-9660 image.

Preparing your application disc

You must include a few important items on your application disc:

- a) **devs:** directory. To avoid the "Workbench 1.3 Copyright Commodore" message at boot time, include a system configuration file in the devs directory with the four workbench colors all set to black.
- b) **c:** directory. Copy the files "rmtm" and "bookit" from the c: directory of the ISODevPak diskette to the c: directory of your application disc.
- c) **root** directory. Copy the file CDTV.TM from the ISODevPak diskette to the root directory of your application disc.
- d) **s:** startup-sequence. Add the command "rmtm" to your startup-sequence in the s: directory of your disk. (If you include the Setpatch command, put rmtm just after Setpatch.)

You should also place the *bookit* command in your startup-sequence. *Bookit* reads the Preferences settings (including palette, screen centering, and other information) from the non-volatile

RAM (NVR) of the CDTV unit. The arguments of the *bookit* command specify the settings to read.

The best place to put *bookit* is the very first line in the startup-sequence. We recommend that you use *bookit bv* in the first line of the startup-sequence so that you will change all of the Workbench colors to black, erase the pointer and center the screen while the trademark image is still on-screen. Then, when *rmtm* removes the trademark image, the user will be looking at a black screen rather than being surprised by the blue-and-white Workbench screen.

ISO Control File

The ISO Control File Builder automatically scans your AmigaDOS drive and builds a control file for the actual CD image builder. This control file is used by BuildTrack to build a CD-ROM track image on an AmigaDOS hard disk or block-for-block disk image on a SCSI disk.

The Control File Builder will build your new ISO file based on the pathname you provide. If, as an example, you wish to build the entire drive, you would simply type ISO <drive name:>. If you want to build only a directory, just give the full path that you wish to build, e.g., "ISO dh0:mydirectory". Always keep in mind that the Control File Builder will build an exact image of an ISO CD. The Control File Builder will construct the ISO directory for you and will optimize that directory structure for you.

Refer to the readme file on the ISO DevPak disk for details on using the *iso* command.

Optimizing the controlfile.

The controlfile is ASCII text, so you can read it or edit it if you wish. You may also wish to relocate where files and/or directories reside on the ISO CD-ROM to optimize performance. The *ISO* utility can sort files by file size (default), file name, or keep the files in the same order as on your source AmigaDOS hard disk. If your application could operate faster with the files in some other order, you are free to change the order of the files.

For example, if file Pic.1 is to be loaded and then the sound files Noise.1, Noise.2 and Noise.3 are to be played sequentially, you might choose to put all four files in one directory, in that order.

What you must *not* do is change the directory in which a file resides. If you wish to do so, you must physically change it on your source hard disk, then run *ISO* again. If you move a file from one directory to another manually, the *BuildTrack* program will return a "file not found" error.

Creating An ISO Image

Two options are available for creating ISO images: a single track image, or a block-for-block disk image. First let's briefly review the format of an ISO disc.

ISO Disc Structure

An ISO-9660 standard CD-ROM is divided up into a number of variable-length tracks. These tracks are recorded in a spiral from the center of the CD outwards. The spiral is three miles long on a full disc.

Track 0, known as the “subchannel”, is reserved for the system. It contains information such as:

- the type of system on which the disc can boot
- the Table of Contents of the disc (TOC). The TOC describes the number of tracks on the disc, their location on the disc, and if the track contains CD audio or data.

Track 1 is the usual location for all data files. Here are stored your program’s code, graphic files, IFF sound files, animations, etc. Again, this track is as long as needed to contain all your application’s data.

Tracks 2–99 are for CD-DA or other data tracks. These tracks are also of variable length. You may determine how to store your CD-DA information on these tracks. You may put one “song” or voice narration per track. Or you can have one long track with numerous samples. The CDTV Device Driver provides the means for playing an entire track, or a certain portion of a track, beginning at a precise location (mm:ss:ff, where mm is minutes, ss is seconds, and ff is frames [75 frames per second]).

Preparing the Hard Disk

If you wish to execute a low-level write of the ISO-9660 image on a SCSI hard disk drive, the *bytedrive* program must be used. Do not perform this step if you are writing the ISO-9660 image to an AmigaDOS file.

A Low-level Write Destroys All. The low-level write to your target SCSI hard disk will eliminate the formatting and all of the data on your target disk. *Do this only on a hard disk that you can overwrite.*

The syntax of the *bytedrive* program is:

```
run bytedrive scsi.device <unit number>
```

<unit number>

The SCSI address of the SCSI hard disk drive to write the ISO-9660 image to.

For example, if the SCSI hard disk drive is configured to SCSI address 5:

```
run bytedrive scsi.device 5
```

Note that *bytedrive* requires both your application disk and the target disk to be attached to the same A2091 controller.

Now from the CLI type:

```
mount DR1:
```

This mounts a special volume, *dr1:* which is addressed by *BuildTrack*.

Creating An ISO Image file

The *Buildtrack* utility lets you create an ISO disc image file of one track on the destination device specified in the *ISO* command. The syntax of the *buildtrack* program is:

```
buildtrack <control file> [-w] [-b <buffers>]
```

<control file>

Specifies the name of the control file created by the *ISO* program. This is generally "controlfile".

-w

Suppress warnings related to ISO file naming conventions.

-b <n>

Specifies <n> number of buffers to use. 200 is the suggested value; the default value is 64. You may increase or decrease this value based on the amount of RAM available.

For example, to build the ISO-9660 image based on the control file "controlfile", suppressing file name warnings and using 200 buffers:

```
buildtrack controlfile -w -b 200
```

The ISO-9660 image will be created, based on the control file and source path.

This process will require several minutes, depending on the size of the application being pre-mastered.

Running FIXTM

In order for your CDTV application to boot directly, you must include a special file called CDTV.TM in your root directory. This file is provided on the ISO DevPak diskette.

The *fixtm* utility, on the same diskette, is used during pre-mastering to update the ISO image file with key information. You should run *fixtm* when you have finished the *buildtrack* utility.

The syntax for the *fixtm* command is:

```
fixtm [-f<filename>] [-d<scsiunitnumber>] [-q] [-h]
```

-f<filename>

Specifies the AmigaDOS file to be updated. If this option is used, by default no Y/N confirmation prompt will be issued.

-d<scsiunitnumber>

Specifies the SCSI unit where the ISO 9660 image resides.

-q

Generates a Y/N prompt from *fixtm* before updating the ISO-9660 image.

-h

Displays a help message.

For example, to update an ISO-9660 image file called Work:MyCDTVApp/ISOImage, you may enter:

```
fixtm Work:MyCDTVApp/ISOImage
```

In this case, *fixtm* will locate the file ISOImage, and ask you if you want to update the image or not. Alternatively, you may enter the command:

```
fixtm -fWork:MyCDTVApp/ISOImage
```

In this case, no prompt will be generated. This method is recommended if you want to include *fixtm* in a script.

If no filename is specified, *fixtm* will search for an ISO-9660 image on SCSI units 5 and 0. If an ISO-9660 image is located, the prompt:

```
Device <n> contains an ISO image, volume name <volume name>.  
Update this image? (y/n)
```

will appear. Enter:

```
y
```

If an ISO 9660 image is not found on SCSI unit 5 or 0, *fixtm* will prompt:

```
Input the name of AmigaDOS image file to be updated:
```

Enter the filename of the ISO-9660 image file. This should be the destination path entered at the "Volume ISO9660" prompt of the *ISO* program.

At this point, the CD pre-mastering is complete.

Cutting a Test Disc

The ISO image must now be transferred to a pre-mastering facility to cut a test disc.

Media for Data Transfer

Currently two methods exist for physically transferring the image to the disk of the pre-mastering system: sending a SCSI hard disk drive and a tape backup.

Ship the drive

The most direct method is to ship the SCSI drive to the pre-mastering center. This method, while primitive, is fast and convenient. We have found that SCSI drives are surprisingly robust. If properly packaged, they can be sent via DHL, FedEx or equivalent and are rarely damaged en route.

The pre-mastering center simply plugs in your SCSI drive, block reads the data across to the Meridian's drive, and returns the drive to you along with the test disc. You will have to specify how many bytes of data are on the drive.

Tape backup

Alternatively you may back up your application to tape. We have tested the A3070 SCSI tape drives from Commodore, and they work well. Their major limitation: they only accept tapes of 150 Mbytes maximum. If your application goes beyond 150 Mbytes, you will have to split up the image, using the BRU utility.

The A3070 can only be used to store a track image. No software is currently available to transfer the block-for-block ISO image from a SCSI drive to the A3070.

- We have been unsuccessful with Exabyte drives. Our testing has found the hardware to be unreliable. However, Exabyte is one of the most popular formats for European replicators.
- We are investigating solutions to back up to a DAT drive, but have yet to find an inexpensive and reliable system.
- No matter what format you choose, make sure you indicate the total size of the ISO file you send to the pre-mastering company. Otherwise they will not know if they have transferred the entire file to their pre-mastering equipment.

Including CD-DA

Most CD-ROM replication companies will produce a mixed-mode CD-ROM/CD-AUDIO disc. If your title includes CD-Digital Audio (CD-DA), you will need to supply high quality audio tapes to the replicator. Make the best quality recording you can because the digital audio process reproduces a nearly perfect recording—any noise, glitch, snap, crackle and pop will also be perfectly reproduced.

You may submit your audio tracks on almost any medium, and that audio does not need to be digitized. All CD replicators got their start in the audio business, so they have a lot of experience in turning music into digital data. The two best formats are: 3/4" U-Matic PCM format or a DAT sampled at 44.1KHz. These two formats can be digitally transferred with no loss of audio quality. An analog medium such as a cassette, or reel-to-reel tape is also acceptable, but these formats must be digitized, with a possible (but not likely) decrease in sound quality.

A sixty second 1Khz test tone recorded at 0db should be at the beginning of every audio master tape. This tone allows the audio technician to calibrate his equipment to give you the proper audio level on the CD.

If you are using a DAT as a master tape, always remember that you must record at 44.1Khz in order to do a digital transfer. Otherwise the DAT must be re-sampled to the proper sampling rate which takes more time and can degrade the sound quality.

Also, if you are using DAT, remember to use the START ID markers at the beginning of each audio track. This allows the technicians to see exactly where the track begins on the DAT. Try to number the START ID at 2 so that the tracks on the DAT correspond to the tracks on the CD (remember that per the Yellow Book Specification, track 1 may *only* contain CD-ROM data).

Regardless of the media you use for your audio, the replicator will need to know where to place the tracks on the CD. Always give the replicator a track sheet that shows the order of the tracks on the CD, the length in minutes and seconds for each track, and the period of separation between each track (which is called the "gap"). You may also specify a specific location (in CD time) to start a particular track if you have that information listed in the track sheet. Refer to the sample Track Sheet in Appendix A of this article.

A CD-ROM track and a CD-Audio track are required to have a gap of four seconds in CD time as per the Yellow Book Specification. However, the gap between two audio tracks may be of any length. The default gap time between audio tracks is two seconds. you may specify any other length, even *zero* seconds, if you require it. Should you need a different gap time, be sure to note it on the Track Sheet. Refer to the sample Track Sheet in Appendix A of this article.

Testing

Now comes a crucial step—the verification of your test disc, before replication in quantity.

“Gold” Disc

The pre-mastering facility will prepare a write-once CD, sometimes called a Gold Disc. This disc can be used for testing purposes.

You may want to ask for more than one copy of the Gold Disc to test in multiple sites.

Testing

We strongly suggest that you test your application thoroughly. Points to consider:

- Try to run through all possible paths to your code and data.
- Load times. Verify that loading of graphics and audio is properly co-coordinated. Make sure you don't start playing an audio file too soon, or too late.
- Animation/video. Verify your animation or video sequences. If things are too slow, you might consider using alternate compression routines.

In Case Of Trouble

If you find problems, you will have to modify your code, then prepare another ISO image, i.e., repeat the steps in the "ISO Control File" section. Do not pass GO. Do not collect \$200.

However, all is not lost. If you have transferred all your data, IFF sound files, animations, code, and even CD-DA data onto the test disc, and discover timing problems or other bugs in your code, *don't throw away your test disc!* You can modify your code on your A2500 development system, and recompile there, then transfer your code to a floppy diskette.

Now connect an A1011 floppy disk drive to the rear of your CDTV reader, insert your test disc in the CDTV, insert your floppy disk, and boot from the floppy. Your program on the floppy can access the data on the CD. It can play CD-DA tracks on the test disc as well. Thoroughly testing your application in this environment can avoid extra trips to the pre-mastering center for test discs.

Mastering And Replication

Having thoroughly tested your application on a write-once CD, you are ready for the final steps: mastering and replication.

Producing The Master Tapes

While it is possible to create a “master stamper” directly from a write-once CD, most replication centers prefer to receive a 9-track tape for input. This tape is prepared by the pre-mastering facility that made your test disc.

The master tape is an ANSI-labeled, 9 track, 1/2 inch tape containing an image of your applications code and data. The file may be split up over 2-4 reels, if the application cannot reside on one.

Preparing CD-DA Tapes

If your application includes CD-DA audio, separate CD audio tapes will be created. CD Audio master tapes preferably have contiguous files with no files broken up across reels. Each 9-track tape holds approximately 138 Mbytes. You can write multiple files to each tape, but all the files must be contiguous.

Replication

Upon receipt of the master tapes, the replicator begins work. A detailed description of the numerous steps involved is beyond the scope of this document. But here is a brief overview.

1. The tape's image is transferred to a large hard disk system. That system adds extra information, such as the synch pattern, the header, and error detection and correction data. Each sector is thus expanded from 2048 bytes to 2352 bytes.
2. The sectors are transferred to the Laser Beam Recorder. This signal-processing rack and recorder adds error correction and subcode data, and performs the low-level encoding. The subcode data tells the CD-ROM drive where the head is located, independent of the sector data. The Recorder then exposes the master disc, a glass disc coated with a very thin layer of photoresist.
3. The master is developed, and the photoresist which was exposed to the laser beam is etched away, leaving the pits. It is then coated with a thin layer of silver. The master can now be read, and is carefully tested on a special player to ensure quality.
4. A "stamper" is created to actually press the CDs. Nickel is electroplated onto the silver surface of the master. The nickel shell is then separated from the glass master, creating a mirror image of the master.
5. The stamper is placed on a molding machine. Molten polycarbonate is pushed between it and a mold. This polycarbonate is given an aluminum reflective layer, and a protective coating is added. It is now a CD.
6. A label is printed on the CD, either via silk screening or pad printing. The CDs are put through a final quality assurance inspection. Finally, the replicator packages the CD, in a jewel case or a long box, and shrink-wraps the product.

The C-Trac Emulator

The C-Trac emulator (*CTrac*) enables the developer to shorten his development cycle and reduce his pre-mastering costs.

Hardware And Installation

CTrac is described in detail in the "CTrac Emulation System" article in the "Creating CDTV Applications" section of this manual. It is a board that plugs into the slot of an Amiga 3000 or Amiga 2500 and is connected via a ribbon cable to the motherboard of a CDTV unit.

The developer must format a hard disk under the Qwik File System (QFS). The QFS disk will be used to store the ISO-9660 image of your data necessary for the emulator. You can also store regular AmigaDOS files on the QFS drive. The AmigaDOS file system will mount the AFS drive, with

its own file system, and you can use standard AmigaDOS commands to manipulate and access the data. Thus the same physical hard disk can hold both the ISO image and the AmigaDOS data and code.

CD-DA Files

Furthermore, you can store CD-DA data on the AFS drive. The CD-DA files are seen as normal files by the file system.

Emulating a CD

When a special command is typed, the emulator activates. (Your application will be running in the memory of the CDTV unit.) It looks for the s:startup-sequence file on the ISO image of the hard disk, loads that file into the memory of the CDTV, and starts to run.

Whenever the CDTV wants to access the CD, the emulator board intercepts the call. It reads the data from the QFS disk in the Amiga, emulating the seek time of a CD. It also slows down the transfer rate from the disk to the memory of the CDTV, to approximate the 150 Kbytes/second used on the CDTV.

The *CTrac* system allows you to test your application in an environment which provides response times very similar to those of a test disc. It can save you thousands of dollars by avoiding repeated cuts of test discs.

Appendix A

Sample Track Sheet

TITLE: Space Waste		DEVELOPER: Fly by Night Multimedia		
CD TRACK #	ROM/ TRACK NAME	ROM AUDIO	GAP TIME	TIME START
01	Awesome Game	CD-ROM STD	2 SEC	
02	Sound Effects Track	AUDIO	0 SEC.	04:15:00
03	Inna Godda Davida	AUDIO	0 SEC	NA
04	Fly Me To The Moon	AUDIO	0 SEC	NA

Notes:

- Even though it seems obvious always be sure to write down the title and customer name at the top of the Track Sheet.
- In this sample Track Sheet, track 1 contains data per the Yellow Book CD-ROM standards. Also notice that the 2 second post-gap is required (also per the Yellow Book).
- Track 2 contains sound effects that will be played back via absolute Minute/Second/Frame reference (using PlayMSF). The track will begin at exactly 4 minutes 15 seconds CD time. Gap time (except for the mandatory CD-ROM gap) is not relevant since the program is addressing the track starting at an absolute location.

- Track 3 is a soundtrack that is played during the opening sequences by playing the track (using PlayTrack). Absolute location is not needed since the PlayTrack function will start playing at the beginning of the track and will stop when directed by the program. Notice that the gap time is zero. The PlayTrack function will start playing at the beginning of the gap, not the beginning of the music. By using a gap time of zero, the music will start playing immediately instead of after the gap.
- Track 4 is another soundtrack that is played exactly like track 3.
- Be sure to write down the total number of tracks on the bottom of each Track Sheet that you submit. You wouldn't want a replicator to miss a track.

Debugging CDTV Software

This article presents some general techniques for debugging software on the Amiga and CDTV. Before you start programming the Amiga or CDTV, you should read the development guidelines in the introductions of the Addison-Wesley *Amiga ROM Kernel Reference and Hardware* manuals, or the *General Amiga Development Guidelines* and the *Developers Introduction* articles at the beginning of this section. These guidelines contain important rules which are applicable to all Amiga programs, configurations, and operating system releases. Additional symptom-specific information on Amiga programming problem areas can be found in the *Troubleshooting Your Software* article that follows. The most common Amiga programming errors are covered in these articles.

The debugging tools, linker libs, and examples referred to in this article may be found on the 2.0 Native Developer Update disks. If you are a registered developer, these disks should be available from your local developer support organization. If you are not a registered developer, check developer bulletin boards for information on where you can purchase these disks in your area.

Preventing Bugs

The best way to debug software is to prevent bugs in the first place. To help prevent bugs, here are some rules you should always follow when writing Amiga and CDTV software:

1. **Know Your Tools.** Familiarize yourself with the debugging tools that are available. If you are familiar with these tools, it is very simple to check for memory loss, loss of signals, improperly nested Forbids, stack size problems, misuse of `IORequests`, overwriting of allocations, and many other problems. Read the tool docs that come with the 2.0 Native Developer Update tools and familiarize yourself with the tools that are available.
2. **Use *Enforcer* and *Mungwall* while developing your code.** If you are developing on a machine without an MMU, upgrade your machine. If that isn't possible, at least use *Mungwall*. Use of *Mungwall* and *Enforcer* can weed out the kind of memory misuse bugs that make software "flaky". Assembler programmers should test for register misuse with *Scratch*. Software publishers and QA departments should insist that software pass testing with these debugging tools.
3. **Read the latest documentation, autodocs, and include file comments for the functions and structures you are using.** Read development guidelines, troubleshooting guides, and compatibility notes. Use the system software and hardware as it was intended and *documented*.
4. **Always check return values from system functions.** Provide a clean way out and useful messages if something fails.
5. **C Programmers should use function prototypes for all functions whether they be system functions or homemade.** It's a little extra work but saves time in the long run by immediately catching most types of improper function calls (missing arguments, swapped arguments, etc.)

6. Keep a version number in your code, and update the version number whenever changes are made. The 2.0 VERSION command can print out the version of any executable which contains a specially formatted version string. Use the tool *Bumprev* to generate C and assembler include files containing a full version string, with date, for use in your program. Or at least code a minimal version string as follows:

In C:

```
UBYTE *vers="\$VER: programname 36.10";
```

In Asm:

```
vers DC.B '\$VER: programname 36.10',0
```

7. Document the code changes for each version. This can be done manually or by using a document control system such as *RCS* (Fish Disk 451). *RCS* will keep diffs of all changes, and will allow you to recreate previous versions of your code if necessary.
8. Write your code in a modular fashion, and use the highest level system functions which provide the functionality you need. Spaghetti code is hard to modify, hard to maintain, and hard to debug.
9. Test your code! Test on different configurations, under different OS's, under low memory and error conditions, and in conjunction with various watchdog tools. Test your product with *MungWall*, in conjunction with *Enforcer* if possible, to catch uses of null pointers and freed memory.
10. If you want to be able to use the debugging tools on a CDTV system, your CDTV titles should be designed to run comfortably in a 1-meg Chip RAM 68000-based 1.3 OS CDTV, with enough RAM left over to allow booting from a floppy and running of debugging tools. However, CDTV developers should not assume that this is enough. If CDTV starts to ship with 2.0 in ROM, will your title be compatible? And is your title compatible with Fast RAM and accelerator add-ons?
11. Design your CDTV title with memory constraints and memory fragmentation in mind. Your title will not run properly if the memory it needs becomes fragmented. Compatibility and debugging will also be difficult if your title uses all of the memory that is available. While designing your title, think of methods to limit or eliminate repeated large memory allocations and deallocations. If possible, allocate audio buffers and bitmap planes once in a size large enough for all intended uses, and recycle without deallocation and reallocation.

Special CDTV Debugging Problems

CDTV applications are debugged in much the same manner as any Amiga application. However, there are some special differences.

How To Use Debugging Tools with a CD

In general, native debugging is best accomplished by booting your CDTV from floppy with a modified Workbench disk. The modified Workbench disk should start (or allow you to start) the desired debugging tools, assign all logical directories and current directory to CD0:, and then start your application software.

To create a debugging boot floppy for CDTV, make a copy of Workbench, throw out unnecessary fonts and tools (such as Diskcopy, Format, Preferences, Edit, etc.), and replace them with the debugging tools you need.

You will probably need a serial terminal (or parallel printer) connected to your CDTV to monitor and capture remote debugging messages from the various tools. Modify the Workbench disk's startup-sequence to provide the functionality you need. You may want to open a NEWSHELL before the ENDCLI of the startup-sequence.

You might want to create a script to start up a set of debugging tools:

```
TNT                ; trap software errors for debugging info
run >NIL: Wack1.0   ; run simple disassembler
run >NIL: Mungwall   ; munge free memory
;Enforcer          ; uncomment only if you have an MMU
execute openscreen.w ; A wedge into OpenScreen() created with LVO
```

and a script for starting your CDTV application:

```
assign sys: CD0:
assign c: sys:c
assign s: sys:s
assign l: sys:l
assign fonts: sys:fonts
assign devs: sys:devs
assign libs: sys:libs
cd CD0:
My_App
```

Start up your debugging tools, and then start your application. For additional information, you may want to compile a special version of your application software with its own remote—`kprintf()` or `dprintf()`—debugging statements. This special version could be started from a floppy disk. It is sometimes useful to make the entire debugging and application startup automatic, including startup of a tool for disassembling (such as *Wack*), and perhaps an editor for taking notes, and the application itself. With such an automatic floppy boot disk startup, your application will have a good chance of loading into the same memory areas each time so that problem addresses reported on successive test runs will be consistent.

Lack of MMU

The lack of an MMU makes it impossible to use the most powerful and timesaving debugging tool, *Enforcer*, on CDTV. *Enforcer* can catch illegal reads and writes to low memory, non-existent memory, and ROM. In conjunction with *Mungwall*, *Enforcer* can also help to catch use of other unallocated, uninitialized, and already-freed memory. These types of bugs are the most common causes of intermittent and hard-to-reproduce problems.

Applications that only reference the CD as a filesystem disk may be tested with an MMU, *Mungwall*, and *Enforcer* by running the title software on your 68020/MMU or 68030/MMU Amiga development system. If your title needs to access files from CD0:, the CD0: drive of your CDTV can be networked to your development system with *Parnet* by Doug Walker, John Toebes, and Matt Dillon (available on Fish Disk 400). *Parnet* uses a specially modified parallel cable to connect two Amigas for filesystem access. Install the *Parnet* software on your CDTV boot floppy and on your development system. Start the netpnet-server on both machines, and then mount NET:CD0 on your development machine. If necessary, you can assign the name CD0: on your development machine to NET:CD0.

Keep That CD Command Handy. If you need to change current directory to the CD0: drive from your development system, you may need to use the 1.3 CD command.

Alternately, you may be able to install a third-party A500 68030 add-on in your CDTV to provide native MMU debugging capability and additional RAM. This will allow you to test with *Enforcer* in the full CDTV environment. In any case, don't forget to also test your software *often* and *thoroughly* on the base 68000 and 1-meg memory configuration.

Lack of Extra Memory

Debugging tools require memory to run, including the overhead and stack of a CLI process (one per tool), and probably the Chip memory of the Workbench/CLI screen kept open by running tools.

If your title uses all available memory, you will not be able to do much native CDTV debugging without some type of memory add-on. However, if your title does not access special CDTV libraries or exec devices directly, you may be able to debug your problem by running your title on your development system as described above. If you are trying to debug memory allocation failure problems, use tools such as *EatMem* or *Memoration* to reduce the amount of memory available to your application.

Personal Memory Cards. You may use the Personal Memory card for CDTV to add 64K or 256K of extra RAM to your CDTV for debugging purposes. This extra memory can be used to hold the debugging tools—*Mungwall*, *TNT*, etc.—that otherwise might not fit. To add a 256K Personal Memory card as extra RAM, use this command:

```
addmem e00100 e3ffff
```

Workbench May Be Inaccessible

Common `printf()` debugging may not be practical in native debugging of a CDTV title because the output window for the debugging text may be inaccessible. Use remote debugging tools whenever possible. Some `printf()` tools, if used individually, may be redirected to SER: or PAR: for remote output. See the information below on remote serial and parallel debugging. In some cases, you may want to redirect your application's own debugging output to your own display. One way to do this would be to write your own debugging output function which would use `sprintf()` or the `exec` function `RawDoFmt()` to format the output, and then `Move()/Text()` the output characters to the `RastPort` of your choice.

No Information When Crashing

When a crash occurs, CDTV reboots instead of putting up an alert. To gain more information about crashes, use the *TNT* tool in your startup. *TNT* installs a trap handler and will try to put up a large debugging information requester if a processor exception occurs. This is valuable when debugging on any system.

TNT Bombs With Debuggers. *TNT* is incompatible with trap-based single-step debuggers, so you must turn off *TNT* before using such a debugger.

General Debugging Concepts

It is hard to generalize about debugging because different kinds of bugs often require very different approaches. A bug report from a user is quite different from a bug that you've just introduced in new code! However, all debugging requires some common steps:

1. Define the problem.
2. Narrow the search and find the bug.
3. Understand, and fix the bug.
4. Make sure you didn't just break something else.

Steps 3 and 4 are the same for all types of bugs, so we'll cover those last. Steps 1 and 2 require different approaches for different kinds of bugs. Here are some examples.

You've added or written new code and something is broken.

1. Define the problem.

Make sure you can reproduce the problem so you'll know when it's gone. Define as "when I do xxx the program does (or doesn't do) do yyy." You might want to write this down in case you get sidetracked while working on your code.

2. Narrow the Search.

If you just added a couple of lines of code, and have the same development environment as before, check your source code first. Check for misuse of existing variables, improper error checking, improper use of system or internal functions, and possible changes to conditional program flow.

If you can't spot the problem, slow it down and see what's going on. Use a source-level or symbolic debugger, or `printf()`/`kprintf()`/`dprintf()` debugging, with delays added if necessary. One particularly useful type of debugging statement is:

```
printf("About to do xxx. k=%ld Ptr1=%lx...\n",k,Ptr1);
Delay(50);
```

The delay gives the debugging line time to be output and gives you a chance to read it before the action is taken. See *mydebug.h* from the 2.0 Native Developer Update debugging examples for an easy way to add debugging statements like this to your code in a neat conditional manner. If you can't `printf()` while your application is running, then link with *debug.lib* or *ddebug.lib* to use the serial `kprintf()` or parallel `dprintf()`.

By stepping through or printing out your actions and variables, you will generally be able to isolate the bug. If you have isolated the area but still can't find the bug, re-read the autodocs for the routines you are using and re-read the *Troubleshooting Your Software* article. Check all other uses of the variables in the problem area. If all else fails, isolate the problem code by writing the smallest possible example that demonstrates the problem.

If the problem is not present in the smallest possible example, then go back and check your original code. Check for global and local variables with the same name. Check for possible overwriting of important program variables, especially those preceded by writable arrays.

If your code has multiple tasks or processes, you may be misusing signal-related structures and functions. Remember that signals are task-relative. One task cannot `Wait()` (or `WaitIO()` or `WaitPort()`, or `DoIO()`) on a `MsgPort` or `Message` whose port signal was allocated by a different task. Such incorrect code may accidentally work if the desired message is already sitting at the port, but don't count on it.

If you still can't isolate the problem, contact CATS for assistance or upload the example to BIX.

BIX Is Quick. One of the quickest ways to find bugs in a small source code example is to upload the source to BIX amiga.dev/main and ask what's wrong with it.

Your code has intermittent problems that you can't pin down, or appears to trash something under certain conditions.

1. Define the problem.

It is difficult to reproduce intermittent problems, so try to force the problem to show itself. First try running your program with *Enforcer* and *Mungwall*. If you don't have an MMU, use *WatchMem* and *Mungwall*, but be prepared to crash a lot. If you don't get any hits, try the same thing during low-memory situations, heavy multitasking and device I/O, etc. If you are doing Exec device I/O, try *IO_Torture* to catch premature reuse of `IORequests`. Hopefully, you will pick up a hit.

2. Narrow the Search.

If you have no *Mungwall/Enforcer* hits, try some debugging statements or source-level debugging to follow the values of your variables. Make sure that all of your structures are cleared before use. Use *TStat* to see if your stack usage is high. Check all possible areas where you might be overwriting the end of an array or otherwise trashing memory. Re-read the autodocs for the system functions you are using.

If you are reusing an *IORequest* too soon, check your source code (debugging would just slow down your execution and might give the *IORequest* a chance to complete, masking the problem).

If you have *Enforcer* hits, use debugging statements or a debugger to step through your code *while* running *Mungwall* and *Enforcer* (or *WatchMem*). This will allow you to pinpoint where the problem occurs.

Your code works fine on one system but not on another. Or you've received a bug report from a user.

1. Define the problem.

Determine what is different or special about the system your program fails on. Remember that if a floppy drive is attached, less free memory is available. Or the difference may be that some type of add-on accelerator board, different chip revisions or different OS ROMs is present. Other considerations include memory configuration and addresses, amount of free Chip and Fast RAM, processor type, custom chip version, expansion peripherals, OS version, and other software in use when the problem occurred. The *ShowConfig* program on the 2.0 Install and 2.0 Native Developer Update disks is useful for printing out much of this information.

The memory address ranges can be particularly important now that some Amigas and add-ons are available with memory beyond the original 24 bit address limit. For example, overwriting a byte array by one byte now has a good chance of trashing a 32-bit address variable, or even your routine's return address on the stack.

If a user reports a problem, find out the exact version of your software she is running, the exact configuration of her hardware, and exactly what she was doing prior to the problem. Try to get her to come up with a step-by-step procedure for reproducing the problem. Get her phone number and keep it with a record of all of the information you can get on the problem. Keep bug reports in an organized form. If you get two reports on the same problem, you can be pretty sure that the problem really exists, and the combined information may help you track it down.

2. Narrow the Search.

Attempt to reproduce the problem. If you can't reproduce it immediately, try stepping through the problem area while using *Enforcer* and *Mungwall*. If you don't get any hits, try again with less free memory and other tasks running. Try to reproduce the user's configuration and environment. If you still cannot reproduce the problem, ask the user to come up with a simple repeatable sequence which causes the problem on a stock system.

Read *Troubleshooting Your Software* for information on the causes of many problems that only show up in certain configurations or environments.

If all else fails, look carefully at your code for misuse of variables or system functions, and for improper error-checking or cleanup after any allocation or open. Check that all cleanups are done in the proper order, i.e., in reverse order.

D. Your program loses memory.

1. Define the problem.

First make sure that you are actually losing memory. Use *Flush* (from the 2.0 Native Developer Update disks) and the system command *Avail* to check for actual memory loss. Under 2.0, you may combine these two by using the *FLUSH* option with the *Avail* command.

Set up your system so you have a shell window available and can start your program without moving any windows (re-arranging windows causes memory fluctuations). Test for memory loss as follows. First, run *Flush* and *Avail* (or *Avail FLUSH*) a few times to make sure nothing else in your system is causing memory to fluctuate. Once you consistently get the same memory values, perform the following steps.

1. *Flush*.
2. *Avail* (write down the Fast, Chip, and total memory free).
3. Start your program and use its features.
4. Exit your program.
5. *Flush*.
6. *Avail* (the fast, chip, and total free to previous figures).
7. If you have a loss, repeat the procedure to see if the loss is consistent.

Again, you can combine *Flush* and *Avail* under 2.0 as *Avail FLUSH*.

Testing with *Flush* or *Avail FLUSH* will flush out all properly closed devices, libraries, and fonts which have been loaded from disk by your program and other programs. This allows you to check for actual memory loss.

Note that under 2.0, since the *audio.device* is ROM-resident but not initialized by the system until it is opened by someone, the first program to use the *audio.device* or speech capabilities will appear to cause a small but permanent memory loss. This is the memory allocated for the audio device's base structures. If your program uses audio or speech, first use the *Say* program or *SPEECH*: before performing the above memory loss test so that the *audio.device*'s initial memory usage will not interfere with your tests.

One special memory loss problem is a continual loss of memory while a program is running. This is generally caused by not keeping up with *IntuiMessages* or not freeing *Locks*.

2. Narrow the search.

Try the above test again, but this time just start your program and exit immediately. If you do not lose memory, try several times more, using some of your program's features, and attempt to determine which part of your program causes the memory loss. Check your source code for all opens and allocations, and check for matching frees and closes, in the proper order, for each of them.

The size of a memory loss can also be a clue to the cause. For example, a loss of exactly twenty-four bytes is probably a `Lock()` which has not been `UnLock()`'d. Knowing the exact size of the loss (as determined with *Flush* and *Avail*) is important when you try determine which allocation is not being freed. The 2.0 Native Developer Update disks and the Addison-Wesley *Amiga ROM Kernel Reference Manual: Includes and Autodocs* both contain a Structure Reference chart that lists the size of each system structure.

Some additional tools on the 2.0 Native Developer Update disks can help determine where memory losses occur. You can use *MemMon* to record the relative memory usage as you test various parts of your program. *Snoop* can be used to record all memory allocations and frees on a remote terminal, after which *SnoopStrip* can strip out all matching pairs. *Mungwall* contains an enhanced snoop option with *SnoopStrip*-compatible output for tracking only the memory allocations of a particular task or tasks. *MemList*, which outputs the system memory list, can also be useful when debugging memory loss and fragmentation.

The *Wedge* program, which can restrict its reporting to the function calls made by a single task or list of tasks, can also be used to monitor the allocations and frees done by your task (however, *Wedge*'s output is not *SnoopStrip*-compatible). By inserting debugging statements in your code, you can mix status messages:

```
About to do xxx with Wedge
```

or

```
Mungwall SNOOP output
```

Examine the output for an allocation which matches the size of your loss. Use *LVO*'s *WEDGELINE* option to generate command lines for *Wedge*.

Removing Bugs

Earlier it was mentioned that there were four basic steps to debugging:

1. Define the problem
2. Narrow the search and find the bug
3. Understand, and fix the bug

4. Make sure you didn't just break something else

Steps 1 and 2 have been covered individually for various types of bugs. Here are steps 3 and 4 for all bugs. This is the easy part (finding the bug is the hard part). These steps are the same for most debugging problems.

3. Understand, and Fix the Bug.

When you find the bug, make sure you understand it. Don't just try something else. If you are having a problem with a system routine, read the autodocs and chapter text for that routine. Consult the *Troubleshooting Your Software* article for similar problems.

When you understand what is wrong, fix the problem, being especially careful not to affect the behavior of any other parts of your program. Carefully document the changes that you make and bump the revision number of the program. Note your changes in the initial comments of the program, and in the area where the changes were made.

4. Make Sure You Didn't Break Anything New.

Try to reproduce the problem several times and make sure it is gone. Thoroughly test the rest of your program and make sure that nothing else has been broken by your fix. Test your program in combination with watchdog tools such as *Mungwall* and *Enforcer*.

Debugging Tools

This section will summarize the purpose and use of various types of debugging tools. Consult the debugging tool docs on the 2.0 Native Developer Update disks for additional tools and documentation.

1. Enforcer, Mungwall, and Other Watchdog and Stressing Tools

Watchdog and stressing tools can alert you to hidden and intermittent problems in your code. The MMU-based Amiga debugging tool, *Enforcer*, provides debugging and quality assurance capabilities far beyond what was previously possible. It is now possible to find bugs even in code that appears to be working perfectly—the kinds of bugs that could cause serious problems on different configurations. *Enforcer* is able to trap improper low memory accesses, writes to ROM, and accesses of non-existent memory—problems which are generally caused by use of freed or improperly initialized pointers or structures.

When used in conjunction with a free memory invalidation tool such as *Mungwall*, additional illegal memory uses are forced out into areas trappable by *Enforcer*.

Another extremely useful testing tool, especially for assembler programmers, is *Scratch* by Bill Hawes. One of the most surprising compatibility problems seen is improper use or dependence on scratch registers (D1,A0,A1) after a system call. *Scratch* allows you to invalidate the contents of these scratch registers after system calls so that improper usage of these registers in your code may be brought out.

It is also useful to test your software with stressing tools such as *EatMem*, *Memoration*, and *EatCycles* in conjunction with *Enforcer* because *Enforcer* will help to catch use of unsuccessful allocations immediately.

All software should be tested with these tools during development, and should be required to pass a test with *Enforcer* in conjunction with *Mungwall*, *Scratch* and *IO_Torture* before being released and distributed.

2. Symbolic and source-level debuggers

Symbolic debuggers allow you to trace and single step through your code, and examine or change your variables and structures. The source-level debuggers which are provided with some compilers allow you to trace and single step your code at the source-level after compiling with special flags. Debuggers can often be used in combination with other tools such as *Mungwall* and *Enforcer* to detect exactly where a problem is occurring.

3. Printf(), kprintf() serial, and dprintf() parallel debugging

This simple method of debugging allows you to monitor where you are, what your variables contain, and anything else you care to print out. **Printf()** debugging is suitable for any process code that is not in a **Forbid()** or **Disable()** (**printf()** breaks a **Forbid()** or **Disable()**). **Kprintf()** (serial) and **dprintf()** (parallel) debugging is more flexible and can be used in process, task, or interrupt code. The **kprintf()** function is provided in the *debug.lib* linker library. The parallel version, **dprintf()**, is provided in the *ddebug.lib* linker library. See the *debug.lib* **kprintf()** autodocs for more information on the types of formats handled by **kprintf()** and **dprintf()**.

Kprintf() outputs to the serial port at the current serial port baud rate. Generally, **kprintf()** is done at 9600 baud with a terminal, or another Amiga running a terminal package connected to your serial port with a null modem serial cable.

However, it is possible to **kprintf()** to yourself (i.e., to a terminal package running on your own machine) if you have a modem attached to your serial port, and your terminal package set to the baud rate of your modem. It is also possible to use a loopback connector to route your machine's serial output to its own input. Obviously, if the problem you are debugging causes you to crash, a remote terminal is a better choice. The ASCII capture feature of your terminal package can be used to capture the **kprintf()** debugging output for later examination.

Remote (**kprintf()**/**dprintf()**) debugging is extremely useful when combined with other remote debugging tools such as *Enforcer* and *IO_Torture* because your own debugging statements will be interspersed with the remote output of the other debugging tools, allowing you to track what your program is doing when problems occur.

Printf()/**kprintf()**/**dprintf()** debugging can be conditionally coded more conveniently by using an include file such as *mydebug.h* (see the DevCon disks). *Mydebug.h* eliminates the need for messy **#ifdef** and **#endif** lines around your debugging statements by providing the conditional macros **D(bug())**, **D2(bug())**, and **DQ(bug())** which take **printf()**-style format strings and arguments in their inner parentheses. One handy feature of these macros is that your debugging statements can be quickly changed from **printf()**s to **kprintf()**s or **dprintf()**s by simply setting a flag in *mydebug.h* and recompiling.

Example:

```
D(bug("I'm here now and a=%ld",a));
```

4. Other ways to debug low-level code

If you can't link with *debug.lib*, low level code can also be debugged by inserting visual or audio cues to let you know where you are. *DebTones.asm* (in AmigaMail Volume 1, September/October 1989 and on the 2.0 Native Developer Update) demonstrates a small audio tone macro suitable for debugging low level code. Another common method is flashing the power LED (see *toGl_led.asm*), or doing an Intuition *DisplayBeep()* to flash the screen.

5. Specialized debugging tools

A variety of specialized debugging tools are available for monitoring and debugging such things as system function calls, device I/O, process status, memory usage, and software errors. These tools can be used without recompiling your program and can provide valuable debugging information. The usage of several prominent tools will be summarized here. See the debugging tool docs on the 2.0 Native Developer Update disks for additional information and usage of these and other tools.

IO_Torture and IO_Torture.par

IO_Torture is used to check for premature re-use of outstanding *IORequests*. This is especially useful for checking any code which does asynchronous device I/O.

Devmon

Devmon allows you to monitor device I/O at the exec level. This can be very useful when debugging your own device code, or code that uses any other exec device.

Wedge

Wedge allows you to wedge into almost any system function and monitor the calls to that function by all tasks or a selected list of tasks. If you know how to use *Wedge*, it can sometimes be much quicker to do *Wedge* debugging than to add your own debugging statements and recompile. You can easily answer questions like "Was that library open successful?" and "Was that file found?" by wedging functions like *OpenLibrary()* and *Open()*. *Wedge* command line arguments are complex—do not attempt to write a *Wedge* command line yourself. Instead, use the *LVO* command's *WEDGLINE* option to generate a *Wedge* command line for the desired function.

LVO

LVO is a multipurpose tool. It requires the label FD: assigned to a directory containing the Amiga FD files (also available on the 2.0 Native Developer Update). *LVO* can be used to list LVO offsets, list relevant FD file lines, generate command lines for *Wedge*, and even guess at which system function contains a particular ROM address.

Example *Wedge* command line generation (redirected to a file):

```
LVO >openlibrary.w exec OpenLibrary WEDGE LINE
```

You may want to edit the generated *Wedge* command line file before executing it as a script. By default, the command line will be set up so that the memory pointed at by all address register arguments will be displayed by *Wedge*. Since some DOS functions use a data register for address arguments (such as the name of a file being opened), you will probably want to modify the *LVO*-generated *Wedge* line for such DOS functions so that the memory pointed at by D1 (the filename in this case) will be displayed. To do this, edit the third hex mask (PTRS) so that the second bit from the right is on (i.e., change 0x8000 to 0x8002). You only want to see the memory pointed at by actual address arguments. See the *Wedge* docs for more details.

LVO's new ROMADDRESS option can be very useful when you are crashing or getting *Enforcer* hits at a ROM address. This new option is used in combination with the *Owner* tool. Since ROM addresses can be very different even on different models running the same version of Kickstart, you *must* do the following two steps on the machine where your problem is occurring.

- First use *Owner* to determine which ROM library module "owns" the address.
- Then use *LVO*'s ROMADDRESS option to find out what function entry of that library is closest to that address. This can help you determine what kind of bad structure or pointer you are passing to cause the crash or hit in ROM.

Memoration (by Bill Hawes)

Memoration allows you to selectively deny memory allocations to a particular task. This allows you to thoroughly test the failure path at every point in your program.

Scratch (by Bill Hawes)

Scratch allows assembler programmers to test their code for misuse of scratch registers D1, A0, and A1 after system calls. *Scratch* thoroughly trashes the contents of D1, A0 and A1 after system calls to catch assembler application code that is improperly using or failing to refresh these registers. Such improper register use by assembler code is a primary cause of compatibility problems.

TStat

TStat allows you to monitor the PC, registers, signals, and stack usage of another task. *TStat* grabs the saved taskswitch-time state of another task and displays it in a shell window. The -tickdelay option causes *TStat* to repeatedly examine the state of the other task every tickdelay 50ths of a second. You can use *TStat*'s new NOCTRL option (for no control characters in the output) and redirect *TStat* to a file, PAR:, or SER:. *TStat* is useful when checking for unfreed signals, mismatched Forbids()/Permits(), and stack usage problems.

To monitor *MyProgram* constantly to the serial port ?,

```
TStat >SER: MyProgram NOCTRL -0
```

How to Use MMU Watchdogs and Other Remote Debugging Tools

Setup for Serial Debugging

Hardware:

When hooking two Amigas together, use a straight RS-232 cable with a null-modem adaptor, or use a null-modem cable. When hooking up an Amiga to another type of computer or terminal, you may or may not need the null-modem (crossed lines) depending on whether the other machine's RS-232 port is designed to be basically a sender or a receiver. Avoid connecting lines which are not directly related to RS-232 because different computers have various power supplies and grounds on these other lines. One type of null modem debugging cable is wired as follows:

Amiga		Terminal
1	—	1
2	—	3
3	—	2
7	—	7
5,6,8	—	20
20	—	8

Software:

For remote debugging at 9600 baud, set the sending machine's Preferences to 9600 baud, and use a 9600 baud terminal or an Amiga running a 9600 baud terminal package (preferably with ASCII capture capability) as the receiving machine. Note that other baud rates can also be used for most serial debugging because normal serial `kprintf()`s do not modify the serial SERPER register and are therefore output at the last baud rate your serial hardware was set to. Test your setup by copying a small text file to SER: or try the *ktest* program from the 1900 DevCon disks. The output should show up on the remote terminal.

Applications can output serial debugging statements by using `kprintf()` from C or `KPrintF` from assembler and linking with *amiga.lib* and *debug.lib*. See the *debug.lib* autodocs for more information. Serial input functions are also available. Also see the *mydebug.h* conditional debugging macros on the 1991 DevCon disks.

Serial Watchdog software setup:

Make sure your test machine is set to the same baud rate as the remote terminal you are connected to. Turn on the ASCII capture of your remote terminal.

For machines with 68030 or 68020+MMU:

```
[RUN] Mungwall (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
Enforcer ON
```

For non-MMU machines (warning—encourages bad software to crash!)

```
[RUN] Mungwall (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
[RUN] WatchMem
```

Local Serial Debugging

Hardware:

If you have a modem attached to your serial port, it is possible to capture your own serial debugging output locally. This setup can be useful as long as the problem you are debugging is not one which crashes the machine.

Software:

Run a terminal package at your modem's baud rate to capture the `kprintf()`s. You probably won't be able to test this setup by copying a file to SER: (since the terminal package probably has an exclusive open on the serial device). Instead, use a small program like *ktest* (on the 1991 DevCon disks) to test your setup, or, if you already have an MMU watchdog installed, try an illegal memory accessor such as *Lawbreaker*. Use the terminal package's ASCII capture feature to capture your debugging output.

Serial Watchdog software setup:

Same as for remote serial debugging, but first start up a terminal package on the test machine, at the baud rate of the attached modem, with ASCII capture turned on.

Setup for Parallel Debugging

Hardware:

To set up for parallel debugging, attach a parallel printer to the Amiga's parallel port and turn the printer on.

Connect Or Hang. If no device is attached to the parallel port, parallel debugging statements will hang waiting for the port hardware.

Software:

Some debugging commands have options for parallel rather than serial output. Examples include *Enforcer.par*, *Mungwall.par*, *IO_Torture.par*, and *Wedge* with the 'p' option. Also, you can send your own debugging statements to the parallel port by using `dprintf()` from C, or `DPutFmt` from assembler, and linking with *ddebug.lib* and *amiga.lib*. On the 1991 DevCon disks, see *dtest.asm* for an example of calling `DPutFmt` from assembler, and *mydebug.h* for debugging macros which can use `printf()`, `kprintf()`, or `dprintf()`.

Parallel Watchdog software setup:

For machines with 68030 or 68020+MMU:

```
[RUN] Mungwall.par (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture.par
Enforcer.par ON
```

For non-MMU machines (warning—encourages bad software to crash!)

```
[RUN] Mungwall.par (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture.par
[RUN] WatchMem
```



Troubleshooting Your Software

Many Amiga programming errors have classic symptoms. This guide will help you to eliminate or avoid these problems in your software.

Audio—Corrupted Samples

The bit data for audio samples *must* be in Chip RAM. Check your compiler manual for directives or flags which will place your audio sample data in Chip RAM. Or dynamically allocate Chip RAM and copy or load the audio sample there.

Character Input/Output Problems

RAWKEY users must be aware that RAWKEY codes can be different letters or symbols on national keyboards. If you need to use RAWKEY, run the codes through `RawKeyConvert()` (see the “Intuition” chapter of the *Amiga ROM Kernel Reference Manual: Libraries & Devices*) to get the proper translation to correct ASCII codes.

Improper display or processing of high-ASCII international characters can be caused by incorrect `tolower()`/`toupper()` conversions, or by sign extension of character values when switched on or assigned into larger size variables. Use unsigned variables such as `UBYTE` (not `char`) for strings and characters whenever possible. Internationally correct string functions are provided in the 2.0 *utility.library*.

CLI Error Message Problems

Improper error messages are caused by calling `exit(n)` with an invalid or missing return value *n*. Assembler programmers using startup code should jump to the startup code's `_exit` with a valid return value on the stack. Programs without startup code should return with a valid value in `D0`. Valid return values such as `RETURN_OK`, `RETURN_WARN`, `RETURN_FAIL` are defined in `<dos/dos.h>` and `<dos/dos.i>`. Values outside of these ranges (-1 for instance) can cause invalid CLI error messages such as “not an object module”.

Useful Hint. If your program is called from a script, your valid return value can be conditionally branched on in the script (i.e., call program, then perform actions based on `IF WARN` or `IF NOT WARN`). `RETURN_FAIL` will cause the script to stop if a normal `FAILAT` value is being used in script.

CLI Won't Close on RUN

A CLI can't close if a program has a **Lock** on the CLI input or output stream ("*"). If your program is `RUN >NIL:` from a CLI, that CLI should be able to close unless your code or your compiler's startup code explicitly opens "*".

Crashes and Memory Corruption

Memory corruption, address errors and illegal instruction errors are generally caused by use of an uninitialized, incorrectly initialized, or already freed/closed pointer or memory. You may be using the pointer directly, or it may be one that you placed (or forgot to place) in a structure passed to system calls. Or you may be overwriting one of your arrays, or accidentally modifying or incrementing a pointer later used in a free/close type function.

Be sure to test the return of all open/allocation type functions before using the result, and only close/free things that you successfully opened/allocated. Use watchdog/torture utilities such as *Enforcer* and *MungWall* in combination to catch use of uninitialized pointers or freed memory, and other memory misuse problems. Use the debugging tool *TNT* to get additional debugging information instead of a Software Error requester. You may also be overflowing your stack—your compiler's stack checking option may be able to catch this. Cut stack usage by dynamically allocating large structures, buffers and arrays which are currently defined inside your functions.

Corruption or crashes can also be caused by passing wrong or missing arguments to a system call (for example, `SetAPen(3)` or `SetAPen(win,3)` instead of `SetAPen(rp,3)`). C programmers should use function prototypes to catch such errors.

If you use short integers, be sure to explicitly type long constants as long (e.g., `42L`). (For example, with short ints, `1 << 17` may become zero). If corruption is occurring during exit, use `printf()` (or `kprintf()`, etc.) with `Delay(n)` to slow down your cleanup and broadcast each step. A bad pointer that causes a system crash will often be reported as an standard 68xxx processor exception `$00000003` or `4`, or less often a number in the range of `$00000006-B`. Or an Amiga-specific alert number may result. See `<exec/alerts.h>` for Amiga-specific alert numbers. Also see "Crashes—After Exit" below.

Crashes—After Exit

If this only happens when you start your program from Workbench, then you are probably `UnLocking()` one of the `WBStartup` message `wa_Locks` or `UnLocking()` the `Lock` returned from an initial `CurrentDir()` call. If you `CurrentDir()`, save the lock returned initially, and `CurrentDir()` back to it before you exit. Only `UnLock()` locks that *you* created.

If you are crashing from both Workbench and CLI, and you are only crashing *after* exit, then you are probably either freeing/closing something twice, or freeing/closing something you did not actually allocate/open, or you may be leaving an outstanding device I/O request or other wake-up request.

You must `AbortIO()` and `WaitIO()` any outstanding I/O requests before you free things and exit (see the *autodocs* for your device, and for `Exec AbortIO()` and `WaitIO()`). Similar problems can be caused by deleting a subtask that might be in a `WaitTOF()`. Only delete subtasks when you are sure they are in a safe state such as `Wait(0L)`.

Crashes—Subtasks and Interrupts

If part of your code runs on a different stack or the system stack, you must turn off compiler stack-checking options. If part of your code is called directly by the system or by other tasks, you must use long code/long data or use special compiler flags or options to assure that code.

Crashes—Window Related

Be careful not to `CloseWindow()` a window during a `while(msg=GetMsg(...))` loop on that window's port (next `GetMsg()` would be on freed pointer). Also, use `ModifyIDCMP(NULL)` with care, especially if using one port with multiple windows. Be sure to `ClearMenuStrip()` any menus before closing a window, and do not free items such as dynamically allocated gadgets and menus while they are attached to a window. Do not reference an `IntuiMessage`'s `IAddress` field as a structure pointer of any kind before determining it is a structure pointer (this depends on the `Class` of the `IntuiMessage`). If a crash or problem only occurs when opening a window after extended use of your program, check to make sure that your program is properly freeing up signals allocated indirectly by `CreatePort()`, `OpenWindow()` or `ModifyIDCMP()`.

Crashes—Workbench Only

If you are crashing near the first DOS call, either your stack is too small or your startup code did not `GetMsg()` the `WBStartup` message from the process message port. If your program crashes during execution or during your exit procedure only when started from WB and your startup opens no `stdio` window or `NIL`: file handles for WB programs, make sure you are not writing anything to `stdout` (`printf()`, etc.) when started from WB (`argc==0`). See also "Crashes—After Exit" above.

Crashes—Only on a 68000 and 68010

This can be caused by illegal instructions (80000000.00000004) such as new 68020/30/40 instructions or inline 68881/882 code. Usually, though, it is caused by a word or longword access at an odd address. This is legal on the 68020 and above, but will generate an Address Error (80000000.00000003) on a 68000 or 68010. The most common causes are:

- Using uninitialized pointers.
- Using freed memory.
- Using system structures improperly (e.g., referencing into `IntuiMessage->IAddress` as a struct `Gadget *` on a non-`Gadget` message).

Crashes—Only on a 68040

It is very difficult to recover from a bus error because of the instruction pipelining of the 68040. If your program has an "Enforcer hit" (i.e., an illegal reference to memory), the resulting 68040 processor bus error will probably crash the machine. Use *Enforcer* (on an '030) to track down your problems, then correct them.

Device-related Problems

Device-related problems may be caused by:

- An improperly initialized port or I/O Request structures (use `CreatePort()` and `CreateExtIO()`).
- Use of an I/O request that is not large enough (see the device's include files and *autodocs* for information on the required type of I/O request).
- Reuse of an I/O request before it has returned from the device (use the debugging tool *IO_Torture* to catch this).
- Failure to abort and wait for an outstanding device request before exiting.
- Waiting on a signal/port/message allocated by a different task.

Disk Icon Won't Go Away

This occurs when a program leaves a **Lock** on one or more of a disk's files or directories. A memory loss of exactly 24 bytes is usually a **Lock()** which has not been **UnLocked**.

DOS-related Problems

In general, any *dos.library* function which fills in a structure for you (for example, `Examine()`), requires that the structure be longword aligned. In most cases, the only way to insure longword alignment in C is to dynamically allocate the structure. Unless documented otherwise, *dos.library* functions may only be called from a process, not from a task. Also note that a process's `pr_MsgPort` is intended for the exclusive use of *dos.library*. However, the port may be used to receive a Workbench **WBStartup** message as long as the message is `GetMsg()`d from the port before *dos.library* is used.

Fails only on 68020/30

The following programming practices can be the cause of this problem:

- Using the upper byte of addresses as flags.
- Doing signed math on addresses; self-modifying code
- Using the **MOVE SR** assembler instruction (use `Exec GetCC()` instead)
- Software delay loops.
- Assumptions about the order in which asynchronous tasks will finish.

The following differences in 68020/30 can cause problems:

- Data and/or instruction caches must be flushed if data or code is changed by DMA or other non-processor modification.
- Different exception stack frame.

- Interrupt autovectors may be moved by VBR.
- 68020/30 CLR instruction does a single write access unlike the 68000 CLR instruction which does a separate read and write access (this might affect a read-triggered register in I/O space—use MOVE instead).

Fails only on 68000

The following programming practices can be the cause of this problem:

- Software delay loops.
- Word or longword access of an odd address (illegal on the 68000). Note that this can occur under 2.0 if you reference `IntuiMessage->IAddress` as a structure pointer without first determining that the `IntuiMessage`'s Class is defined as having a structure pointer in its `IAddress`.
- Use of the assembler CLR instruction on a hardware register which is triggered by any access. The 68000 CLR instruction performs two accesses (read and write) while 68020/30 CLR does a single write access. Use MOVE instead.
- Assumptions about the order in which asynchronous tasks will finish;
- Use of compiler flags which have generated inline 68881/68882 math coprocessor instructions or 68020/30 specific code.

Fails only on Older ROMs or Older WB

This can be caused by asking for a library version higher than you need *Do not use the #define LIBRARY_VERSION when compiling!*

It can also be caused by calling functions or using structures which do not exist in the older version of the operating system. Ask for the lowest version which provides the functions you need (usually 33), and exit gracefully and informatively if an `OpenLibrary()` fails (returns NULL). Or code conditionally to only use new functions and structures if the available library's `lib->Version` supports them.

Fails only on Newer ROMs or Newer WB

This should not happen with proper programming. Possible causes include:

- Running too close to your stack limits or the memory limits of a base machine (newer versions of the operating system may use slightly more stack space in system calls and usually use more free memory).
- Using system functions improperly.
- Not testing function return values.
- Improper register or condition code handling in assembler code. Remember that result, if any, is returned in D0, and condition codes and D1/A0/A1 are undefined after a system call.
- Using improperly initialized pointers.

- Trashing memory.
- Assuming something (such as a flag) is B if it is not A.
- Failing to initialize formerly reserved structure fields to zero.
- Violating Amiga programming guidelines (for example, depending on or poking private system structures, jumping into ROM or depending on undocumented or unsupported behaviors).
- Failure to read the function *autodocs*.

See the *2.0 Compatibility Problems Areas* article for in-depth information on 2.0 compatibility problem areas.

Fails only on Chip-RAM-Only Machines

This is caused by specifically asking for or requiring MEMF_FAST memory. If you don't need Chip RAM, ask for memory type 0L, or MEMF_CLEAR, or MEMF_PUBLIC|MEMF_CLEAR as applicable. If there is Fast memory available, you will be given Fast memory. If not, you will get Chip RAM. May also be caused by trackdisk-level loading of code or data over important system memory or structures which might reside in low Chip memory on a Chip-RAM-Only machine.

Fails only on machines with Fast RAM

Data and buffers which will be accessed directly by the custom chips *must* be in Chip RAM. This includes bitplanes (use `OpenScreen()` or `AllocRaster()`), audio samples, trackdisk buffers, and the graphic image data for sprites, pointers, bobs, images, gadgets, etc. Use compiler or linker flags to force Chip RAM loading of any initialized data needing to be in Chip RAM or dynamically allocate Chip RAM and copy any initialization data there.

Fails only with Enhanced Chips

This is usually caused by writing or reading addresses past the end of older custom chips, or writing something other than 0 (zero) to bits which are undefined in older chip registers, or failing to mask out undefined bits when interpreting the value read from a chip register.

Note that system copper lists are different under 2.0 when ECS chips are present. See also "Fails only on Chip-RAM-Only Machines".

Fireworks

A dazzling pyrotechnic video display is caused by trashing or freeing a copper list which is in use, or trashing the pointers to the copper list. If you aren't messing with copper lists, see "Crashes and Memory Corruption".

Graphics—Corrupted Images

The bit data for graphic images such as sprites, pointers, bobs, and gadgets *must* be in Chip RAM. Check your compiler manual for directives or flags which will place your graphic image data in Chip RAM. Or dynamically allocate Chip RAM and copy them there.

Hang—One Program Only

Program hangs are generally caused by waiting on the wrong signal bits, on the wrong port, on the wrong message, or on some other event that will never occur. This can occur if the event you are waiting on is not coming, or if one task tries to `Wait()`, `WaitPort()` or `WaitIO()` on a signal, port, or window that was created by a different task. Both `WaitIO()` and `WaitPort()` can call `Wait()`, and you cannot `Wait()` on another task's signals. Hangs can also be caused by verify deadlocks. Be sure to turn off all Intuition `VERIFY` messages (such as `MENUVERIFY`) before calling `AutoRequest()` or doing disk access.

Hang—Whole System

This is generally caused by a `Disable()` without a corresponding `Enable()`. It can also be caused by memory corruption, especially corruption of low memory. See "Crashes and Memory Corruption".

Memory Loss

First determine that your program is actually causing a memory loss. It is important to boot with a standard Workbench because a number of third party items such as background utilities, shells, and network handlers dynamically allocate and free pieces of memory. Open a shell for memory checking, and a shell or Workbench drawer for starting your program. Arrange windows so that all are accessible, and so that no window rearrangement will be needed to run your program.

In the shell, type `Avail FLUSH<RET>` several times (2.0 option). This will flush all non-open disk-loaded fonts, devices, etc., from memory. Write down the amount of free memory. Now without rearranging any windows, start your program and use all of your program features. Exit your program, wait a few seconds, then type `Avail FLUSH<RET>` several times. Write down the amount of free memory. If this matches the first value you wrote down, then your program is fine, and is not causing a memory loss.

If memory was actually lost and your program can be run from CLI or Workbench, then try the above procedure with both methods of starting your program. Note that under 2.0, there will be a slight permanent (until reboot) memory usage of about 672 bytes when the *audio.device* or *narrator.device* is first opened. See "Memory Loss—CLI Only" and "Memory Loss—WB Only" if appropriate.

If you lose memory from both WB and CLI, then check all of the `open/alloc/get/create/lock` type calls in your code, and make sure that there is a matching `close/free/delete/unlock` type call for each of them (note—there are a few system calls that have or require no corresponding free—check the *autodocs*). Generally, the `close/free/delete/unlock` calls should be in opposite order of the allocations.

If you are losing a fixed small amount of memory, look for a structure of that size in the *Structure Offsets* listing in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*. For example, a loss of exactly 24 bytes is probably a `Lock()` which has not been `UnLocked()`.

If you are using `ScrollRaster()`, be aware that `ScrollRaster()` left or right in a Superbitmap window with no `TmpRas` will lose memory under 1.3 (workaround—attach a `TmpRas`). If you lose much more memory when started from Workbench, make sure your program is not using `Exit(n)`. This would bypass startup code cleanups and prevent a Workbench-loaded program from being unloaded. Use `exit(n)` instead.

Memory Loss—CLI Only

Make sure you are testing in a standard environment. Some third-party shells dynamically allocate history buffers or cause other memory fluctuations. Also, if your program executes different code when started from CLI, check that code and its cleanup. And check your *startup.asm* if you wrote your own.

Memory Loss—Ctrl-C Exit Only

This results when you have Amiga-specific resources opened or allocated and you have not disabled your compiler's automatic Ctrl-C handling (causing all of *your* program cleanups to be skipped). Disable the compiler's Ctrl-C handling and handle Ctrl-C (`SIGBREAKF_CTRL_C`) yourself.

Memory Loss—During Execution

A continuing memory loss during execution can be caused by failure to keep up with voluminous IDCMP messages such as `MOUSEMOVE` messages. Intuition cannot re-use IDCMP message blocks until you `ReplyMsg()` them. If your window's allotted message blocks are all in use, new sets will be allocated and not freed till the window is closed. Continuing memory losses can also be caused by a program loop containing an allocation-type call without a corresponding free.

Memory Loss—Workbench Only

Commonly, this is caused by a failure of your code to unload after you exit. Make sure that your code is being linked with a standard correct startup module, and do *not* use the `Exit(n)` function to exit your program. This function will bypass your startup code's cleanup, including its `ReplyMsg()` of the `WBStartup` message (which would signal Workbench to unload your program from memory).

You should exit via either `exit(n)` where *n* is a valid DOS error code such as `RETURN_OK` (`<dos/libraries.h>`), or via final `"}` or `return`. Assembler programmers using startup code can `JMP` to `_exit` with a long return value on stack, or use the RTS instruction.

Menu Problems

A flickering menu is caused by leaving a pixel or more space between menu subitems when designing your menu. Crashing after browsing a menu (looking at menu without selecting any items) is caused by not properly handling MENUNULL select messages. Multiple selection not working is caused by not handling NextSelect properly. See the “Intuition Menus” chapter of the *Amiga ROM Kernel Reference Manual: Libraries and Devices*.

Out-of-Sync Response to Input

This is caused by failing to handle all received signals or all possible messages after a `Wait()` or `WaitPort()` call. More than one event or message may have caused your program to awaken. Check the signals returned by `Wait()/WaitPort()` and act on every one that is set. At ports which may have more than one message (for instance, a window’s IDCMP port), you must handle the messages in a `while(msg=GetMsg(...))` loop.

Performance Loss in Other Processes

This is often caused by a one program doing one or more of the following:

- Busy waiting or polling.
- Running at a higher priority
- Doing lengthy `Forbid()`s, `Disable()`s or interrupts.

Performance Loss—A3000

If your program has “Enforcer hits” (i.e., illegal references to memory caused by improperly initialized pointers), this will cause Bus Errors. The A3000 bus error handler contains a built-in delay to let the bus settle. If you have many enforcer hits, this could substantially slow down your program.

Trackdisk Data not Transferred

Make sure your trackdisk buffers are in Chip RAM under 1.3 and lower versions of the operating system.

Windows—Borders Flicker after Resize

Set the `NOCAREREFRESH` flag. Even `SMART_REFRESH` windows may generate refresh events if there is a sizing gadget. If you don’t have specific code to handle this, you must set the `NOCAREREFRESH` flag. If you do have refresh code, be sure to use the `Begin()/EndRefresh()` calls. Failure to do one or the other will leave Intuition in an intermediate state and slow down operation for all windows on the screen.

Windows—Visual Problems

Many visual problems in windows can be caused by improper font specification or improper setting of gadget flags. See the “2.0 Compatibility Problems Areas” article for in-depth information on common problems.

General Debugging Techniques

Narrow The Search

Use methodical testing procedures and debugging messages if necessary, to locate the problem area. Low level code can be debugged using `kprintf()` serial (or `dprintf()` parallel) messages. Check the initial values, allocation, use, and freeing of all pointers and structures used in the problem area. Check that all of your system and internal function calls pass correct, initialized arguments, and that all possible error returns are checked for and handled.

Isolate the problem

If errors cannot be found, simplify your code to the smallest possible example that still functions. Often you will find that this smallest example will not have the problem. If so, add back the other features of your code until the problem reappears, then debug that section.

Use debugging tools

A variety of debugging tools are available to help locate faulty code. Some of these are source level and other debuggers, crash interceptors, vital watchdog and memory invalidation tools like *Enforcer* and *MungWall*.

A Final Word About Testing

Test your program with memory watchdog and invalidation tools on a wide variety of systems and configurations. Programs with coding errors may appear to work properly on one or more configurations, but may fail or cause fatal problems on another. Make sure that your code is tested on both a 68000 and a 68020/30, on machines with and without Fast RAM, and on machines with and without enhanced chips. Test all of your program functions on every machine.

Test all error and abort code. A program with missing error checks or unsafe cleanup might work fine when all of the items it opens or allocates are available, but may fail fatally when an error or problem is encountered. Try your code with missing files, filenames with spaces, incorrect filenames, cancelled requesters, Ctrl-C, missing libraries or devices, low memory, missing hardware, etc.

Test all of your text input functions with high-ASCII characters (such as the character produced by pressing Alt-F then “A”). Note that RAWKEY codes can be different keyboard characters on national keyboards (higher levels of keyboard input are automatically translated to the proper characters).

If your program will be distributed internationally, support and take advantage of the additional screen lines available on a PAL system. Enhanced Agnus chip machines may be switched to be PAL or NTSC via motherboard jumper J102 in A2000s and jumper J200 in A3000s. Note that a base PAL machine will have less memory free due to the larger display size.

Write good code. Test it. Then make it great.

- b) For very long searches that cannot be done in a short period of time, inform the user of the progress of the search. Options include putting up a screen and start listing "hits" or showing a "gas gauge" depicting the progress of a search. The user should be able to halt a long search in progress, retaining the results found to that point.
- 34. Multimedia elements should be comparable to video or cartoons viewed on TV. These elements (animations, speech, music, sounds, video) should be streamed from disk so that they can be more in-depth and longer in duration. The animations should normally be 3 dimensional and change focus (i.e., background, perspective), not limited to a static background screen.
- 35. Educational titles and adventure type recreational products need to have a depth of interactivity options. For instance, if a character is walking down a street, the user should be able to go down alleyways, into buildings, etc. Each screen or in each section should have more than one (and more than two!) things that can be done. These options should include non-linear choices, i.e., being able to jump around. Linear choices are really no choices at all because you must follow a prescribed path.
- 36. Educational titles should have some type of testing function to allow you to examine your progress in a section. The Bookmark feature should be used if appropriate (e.g., game scores, place in a book, tests, etc.).
- 37. Reference titles should allow numbers and spaces to be input for searches. All reference titles should support searches on keywords in body or title, and not be just an alphabetized index of options (similar to the index of a book). They should also have the Bookmark feature using Non-Volatile RAM (NVR) to save search criteria and possibly the resultant elements.
- 38. Recreational titles should use continuous streamed animations and CD audio for background. They should be able to save game states and high scores using NVR.
- 39. Possible suggestions:
 - a) Online help
 - b) Templates to fit on top of the IR controller to simplify the buttons for complex products (e.g., flight simulator).
 - c) Optionally viewable demo commercials of other products.
 - d) Hardware add-ons (a la Nintendo).
 - e) Supply a formatted disk (or at least a disk label) if the product can use a floppy.

Overview of CDTV And A500 Differences

You can sum up the differences between CDTV and the A500 in two words: extra goodies. CDTV has extras that the A500 does not, and it is those extras that make it more than just an A500 in a sleek, black box. (Of course, it's missing a few things like a keyboard and an internal floppy.)

The obvious difference is the CD drive with its ability to act as a CD-ROM drive, a CD-DA drive, a CD+G drive and a CD+MIDI drive. Initially, the target audience is expected to use CDTV as a CD-ROM device and a CD-DA/CD+G device. As the product evolves, more titles will come out utilizing it as a CD+MIDI device and when the keyboard and floppy drive accessories come into use, its power will really be seen.

Hardware Items

In addition to the CD drive, CDTV has extra front and rear ports, internal slots and Non-volatile RAM (NVR).

Rear Ports

- MIDI IN and OUT
- Infrared Remote

Front Ports

- 256K Memory Card
- Headphone Jack

Internal Slots

- Intelligent video slot
- DMA internal slot

Non-volatile RAM

- 2K of Non-volatile RAM

Additional Rawkey Codes

The CD drive control buttons on the infrared remote are mapped to four additional rawkey codes.

Play/Pause	6C
Stop	6D
Fast Forward	6E
Fast Reverse	6F

Disc Packaging Standards And Graphics Standards



DISC PACKAGING &
GRAPHICS STANDARDS
MANUAL

 **Commodore**

© Copyright 1990
Commodore Electronics Ltd.
All rights reserved.

Commodore, the Commodore logo, CDTV, and CDTV MULTIMEDIA
are trademarks of Commodore Electronics Ltd.

ONLY LICENSED ENTITIES ARE AUTHORIZED TO USE THE CDTV LOGO.

For licensing information, contact:

Gail Wellington
Commodore
1200 Wilson Drive
West Chester, PA 19380
(215) 431-9204

TABLE OF CONTENTS

Introduction	c
How to use this guidebook	d
Help	d
What's in your package?	e
Categories & age groups	g
Section 1: CDTV MULTIMEDIA LOGO	
Control space	1.2
Color	1.3
Unacceptable logo forms	1.4
Section 2: LONG BOX	
Physical format	2.1
Materials	2.1
Inner construction	2.2
Graphic design	2.3
Section 3: WINDOW LONG BOX	
Physical format	3.1
Materials	3.1
Inner construction	3.2
Graphic design	3.3
Section 4: DISC CADDY	
CDTV MULTIMEDIA logo placement	4.1
Section 5: CADDY SLEEVE	
Physical format & graphic design	5.1
Section 6: COVER INSERT	
Physical format & materials	6.1
Graphic design	6.2
Section 7: INLAY CARD	
Physical format & materials	7.1
Graphic design	7.2
Section 8: DISC GRAPHICS	
Graphic design	8.1
Section 9: PROMOTIONAL MATERIALS	
Graphic design	9.1
Section 10: REPRODUCTION PROOFS	
CDTV MULTIMEDIA logo & Compact Disc logo	10.1
Category symbols	10.2
Glossary	
Master Guides	
Long Box Master Guide	Supplement
Window Long Box Master Guide	Provided with
Long Box Background Negative	Manual

INTRODUCTION

SPECIAL NOTE TO THIRD-PARTY PUBLISHERS:

This publication was created to establish a universal standard for the packaging and visual identity of CDTV discs.

Within these pages you will find clear and concise instructions that will guide you in preparing CDTV disc packaging. Detailed diagrams and reproduceable artwork are provided for every element of your package.

Commodore's goal is to create an image that will serve as an optimum marketing environment, not only for CDTV technology, but for each and every title for years to come. We feel we have succeeded in this endeavor.

Naturally, Commodore cannot exercise any authority over independent publishers in the creation of their packaging, nor do we seek to imply any in the publication of these standards.

However, all publishers of CDTV products share a responsibility to one another to participate in presenting a unified image to the public, so that we may all reap the benefits of a quality image.

In that spirit, we respectfully ask for your cooperation in conforming to these standards.

Thank you for supporting CDTV
with your applications,



David Rosen
Director of International Marketing
Commodore International Ltd.

HOW TO USE THIS GUIDEBOOK

Creating CDTV Disc packaging will be quick and simple if you learn to follow the instructions within these pages.

- 1** Make sure you **completely read** the opening section of the book, beginning with the Introduction. This is usually the first thing people skip in any book, but in ours it provides an important basis for understanding the guidebook. Also be sure to read Section 1: CDTV MULTIMEDIA Logo. It gives you the information you need for using the CDTV MULTIMEDIA logo which is the foundation of the entire design.
- 2** Determine the **Category** that your CDTV title falls within and the **Age Group** of its target audience (see page g).
- 3** **Determine the elements** that will make up your packaging (see page e).
- 4** **Refer to the Section** of the book that describes each element you need to create. (See the Table of Contents for help in locating each Section.)
- 5** **Follow the diagrams** and templates found both in each section of the manual and the envelope (labelled "SUPPLEMENT") that accompanies the manual.
- 6** Use the **reproduction proofs** provided in Section 10 in creating your artwork.

THINGS TO KEEP IN MIND:

- ☐ The diagrams shown throughout the book are NOT ALWAYS ACTUAL SIZE. (The size limitations of this publication do not always allow full size diagrams.) Measurements are provided with each diagram. If a diagram is provided actual size, it will be accompanied by the label "ACTUAL SIZE".
- ☐ Reproduction artwork is provided in the last section of the book.

HELP!...

If you have any questions, or seek further guidance in preparing your packaging, please feel free to contact the creators of the guidebook:

Communiqué
30 Galesi Drive
Wayne, New Jersey 07470
(201) 785-9115

WHAT'S IN YOUR PACKAGE?

Your CDTV Disc package can consist of a combination of several different elements. (i.e.: long box, jewel case insert, etc.) . What goes into your package will be determined by several factors, including the nature of the title, your production budget and your aesthetic preferences. Since the packaging elements are interdependent, you should determine the make-up of your package before any production is begun.

DECISIONS, DECISIONS...

Some of the decisions that you'll face in packaging a CDTV title are:

? Will the outer packaging be a standard Long Box or a Window Long Box?

The Window Long Box provides an economical outer package. A window is left open for the Jewel Case or Disc Caddy to show through, which then becomes your title graphics. The outer box can then be printed simply in 2-color.

The standard Long Box will give you more space to employ your title graphics, but will most likely need to be printed in 4-color process, a more expensive method.

▲ COMMODORE RECOMMENDS: Window Long Box.

Not only is it economical but it will help provide a unified image for CDTV titles and set them apart from conventional CD and CD+G packaging.

? Will the disc itself be packaged inside a Jewel Case or a Disc Caddy?

CDTV owners will require Disc Caddies to play CDTV titles. However, inclusion of them with each disc will add to the cost.

▲ COMMODORE RECOMMENDS: Disc Caddy.

The additional value will be appreciated by customers and will translate into a increased perception of you and your products.

? Will there be any additional documentation, reference cards, or other items included in the box?

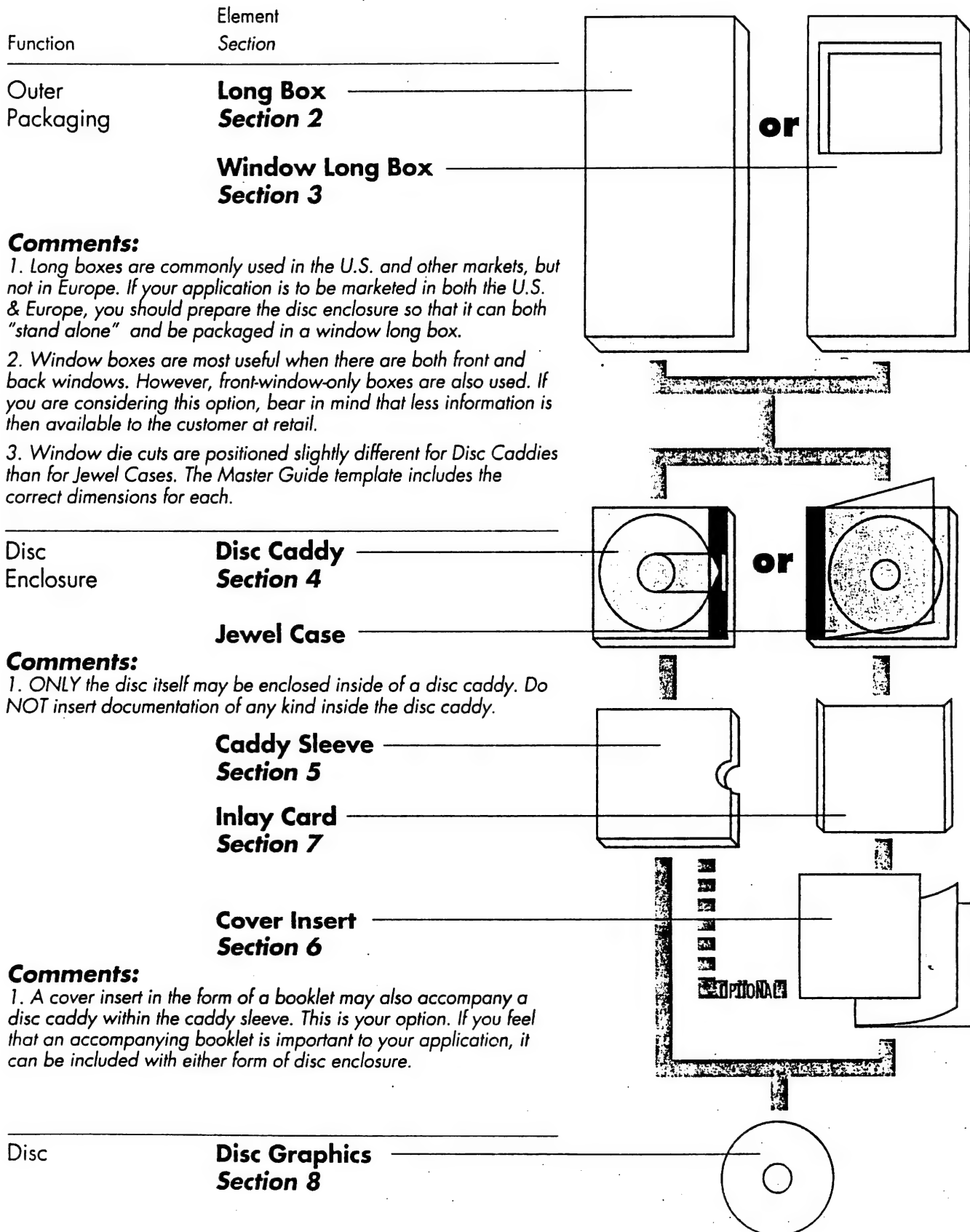
If so, be sure to prepare your outer box with the proper construction to provide enough room.

continued

WHAT'S IN YOUR PACKAGE?

continued

Below is a simple flow chart that outlines your packaging alternatives.



CATEGORIES & AGE GROUPS

CDTV applications will span a broad spectrum of interests and audiences. Because of this diversity, and to help introduce consumers to the world of CDTV, Commodore has devised a system of seven categories to help explain each title and identify its likely audience.

Choosing a specific category for your title may seem difficult and limiting at times. By nature, many CDTV applications cross common borders of specialization. However, we strongly urge that you choose **ONE AND ONLY ONE category and ONE age group** that best applies to your title. Remember, this is only a supplemental aid for customers. Your title graphics and description will tell the real story, so we suggest that you exercise *that* opportunity to present the features and benefits of your application.

The categories are as follows:

<i>Category</i>	<i>Description</i>
Arts & Leisure	Non-game activities for spare time including hobbies, how-to, etc. <i>Examples:</i> gardening, home repair, video tilling, performing and visual arts references
Education	Learn-to or practice basic education skills. Also academic tools. <i>Examples:</i> Learn-to-read, simulations of physics experiments, foreign language tutors
Entertainment	Games of all types (action, thinking, board game simulations, etc.) <i>Examples:</i> Flight simulators, arcade & adventure games, chess
Music	All types of music-oriented titles, including participatory & reference. <i>Examples:</i> Learn to play recorder, classical music reference works, play along, MIDI, CDTV-specific CD+G
Periodicals	Reference or other works released on a regular schedule. <i>Examples:</i> Catalogs, magazines, directories
Productivity	Traditional computer applications. <i>Examples:</i> Word processors, spreadsheets, databases, home accounting
Reference	Sources of information intended to be looked up or explored. <i>Examples:</i> Dictionaries, encyclopedias, atlases, "coffee-table books"

AGE IDENTIFICATION

The age of your target audience should be described in one of the following manners. Again, please choose only one description:

For Children Ages ___ to ___
For Teens & Adults
For All Ages
For Adults

CDTV MULTIMEDIA LOGO

The CDTV MULTIMEDIA logo, based on the original CDTV logo, is the foundation for this entire identity program. Like these guidelines, it is the result of a long, intense creative process to develop a trademark that will project a unique, positive identity for these products.

In the use of the CDTV MULTIMEDIA logo, we ask that you maintain strict adherence to the instructions in this chapter.

LOGO

The CDTV logo consists of the CDTV logotype, the disc graphic, the type "MULTIMEDIA" and the "TM" symbol. **These elements are NEVER to be separated, used independently of each other, or in any way altered from the form shown below, except as specified in this publication.**

THE LOGO IS NEVER TO BE REDRAWN OR RE-CREATED. IT MUST ALWAYS BE REPRODUCED PHOTOMECHANICALLY FROM THE REPRODUCTION PROOFS IN THIS MANUAL (see Section 10 for reproduction proofs.)

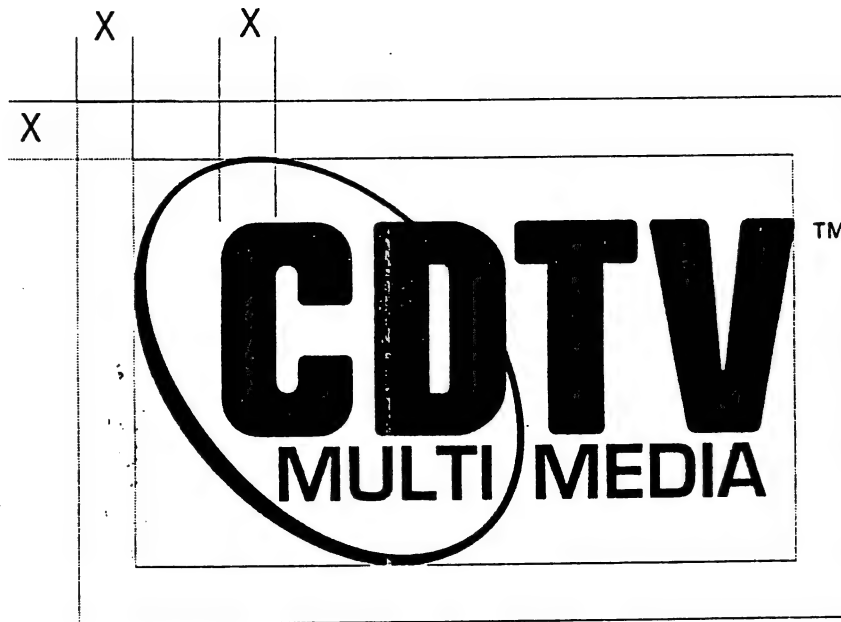


DO NOT REPRODUCE FROM THIS PAGE
(see Section 10 for reproduction proofs.)

CONTROL SPACE

The logo has a minimum requirement of free space around it, with no other graphic elements intruding. This is called the "control space".

The control space is equal to the width of the uprights of the letter "C" in CDTV. In the diagram below it is labelled "X".



DO NOT REPRODUCE FROM THIS PAGE
(see Section 10 for reproduction proofs.)

CDTV MULTIMEDIA LOGO

continued

COLOR

The logo must always appear...

1. In either black ink, or as a white reverse image

ACCEPTABLE:



ACCEPTABLE:



DO NOT REPRODUCE FROM THIS PAGE

(see Section 10 for reproduction proofs.)

UNACCEPTABLE LOGO FORMS



NEVER omit or separate any of the elements.



NEVER alter the letter spacing of the typography.



NEVER alter the typestyle or placement of the typography.



NEVER stack the typography or alter its arrangement.



NEVER place the logo in a box, frame or other graphic.



NEVER place the logo vertically.



NEVER intermix shades or colors.

Section 2: LONG BOX

The long box is the outer packaging used for CD's in many major markets, including the U.S. It has been similarly adopted as a outer packaging for CDTV discs in those same markets.

PHYSICAL FORMAT

The long box, when fully constructed, measures approximately 12.3" tall, by 5.7" wide, by .5" deep. Along with this manual, you should have received a film template labelled CDTV LONG BOX: MASTER GUIDE. (The film is too large for inclusion in this book.)

The Master Guide gives you all the dimensions, diagrams and instructions you will need in creating your long box.

MATERIALS

Commodore recommends and encourages the use of **recycled paper** in the making of long boxes for CDTV discs. Serious environmental concerns have caused intense criticism of long boxes in CD marketing. However, their function as both a theft deterrent and a marketing tool have kept them in use.

We feel that the only environmentally responsible solution to the problem is to use recycled paper in our long boxes.

We further encourage that, when it applies, you include an appropriate statement to that effect on the back of your box.

EXAMPLE:

Made from recycled paper.

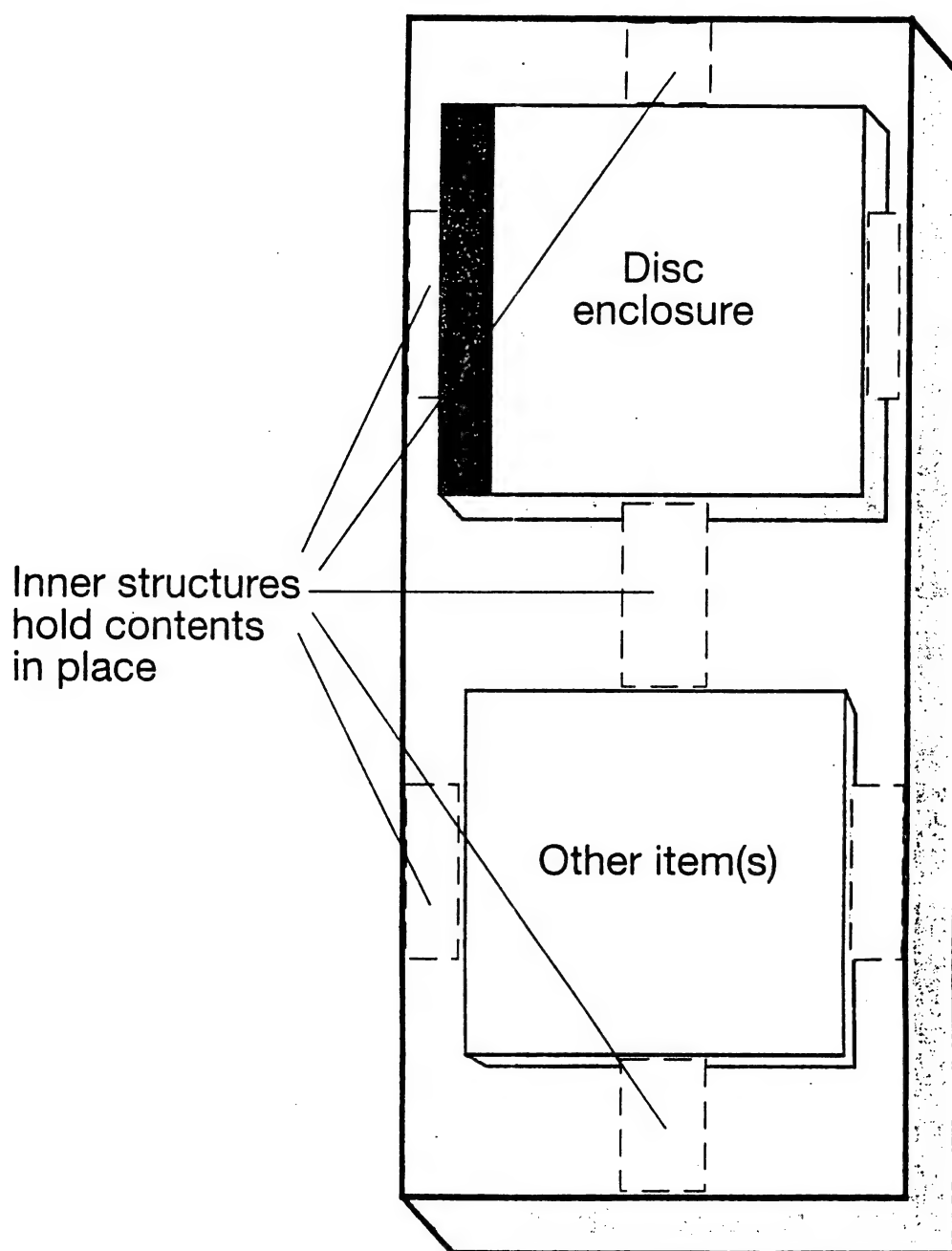
OR, if applicable

Made from 100% recycled paper.

INNER CONSTRUCTION

Each long box must be specially die cut to provide for the proper inner construction which provides both support and strength for the box itself and will hold the contents in place.

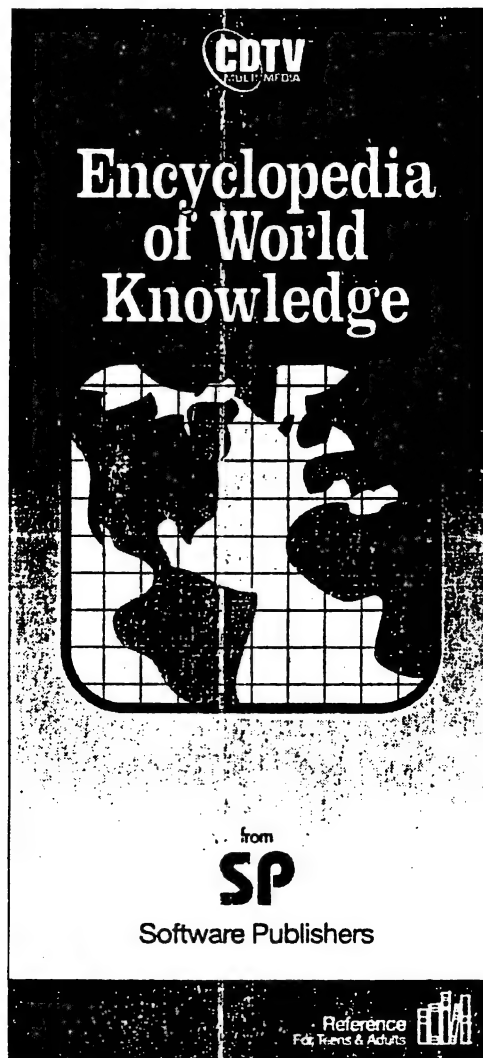
This is of particular importance when additional materials, other than the disc enclosure, is included, such as reference guides, "smart cards", etc.



GRAPHIC DESIGN

With this manual you received a Long Box MASTER GUIDE, which is a large, acetate template. This template shows an actual-size diagram of an unconstructed window long box and complete instructions for creating your artwork.

Below is a depiction of the final result of those instructions. There are areas provided within the design for you to place your own graphics depicting the application title, features, etc., and your company's logo and information. You may "block off" these areas and use the entire space, or simply place your graphics over our background within the areas, in "silhouette" fashion.



FRONT

For all instructions regarding the graphic design of the window long box, refer to the film template labelled CDTV LONG BOX: MASTER GUIDE provided with this manual.

Section 3: WINDOW LONG BOX

The window long box is a modification of the standard CD long box which provides a die-cut opening, or window, in the front & back panels of the box. Through these openings the disc enclosure itself is visible which eliminates the need to re-create it on the exterior of the long box. This is also a useful technique when the same application is to be offered in markets where long boxes are not used. This way, the same disc enclosure can be used in multiple markets (barring language barriers).

Window boxes can also have only one die-cut window in the front, but this, of course, does not permit the back of the disc enclosure to show through.

PHYSICAL FORMAT

The window long box shares the same dimensions as a standard long box, measuring approximately 12.3" tall, by 5.7" wide, by .5" deep. The die-cut window is positioned in the upper half of the box and measures approximately 4.9" x 4.4".

Along with this manual, you received a film template labelled CDTV WINDOW LONG BOX: MASTER GUIDE. (The film is too large for inclusion in this book.)

The Master Guide will give you all the dimensions, diagrams and instructions you will need in creating your long box.

MATERIALS

Commodore recommends and encourages the use of **recycled paper** in the making of long boxes for CDTV discs. Serious environmental concerns have caused intense criticism of long boxes in CD marketing. However, their function as both a theft deterrent and a marketing tool have kept them in use.

We feel that the only environmentally responsible solution to the problem is to use recycled paper in our long boxes.

We further encourage that, when it applies, you include an appropriate statement to that effect on the back of your box.

EXAMPLE: Made from recycled paper.

 OR, if applicable

 Made from 100% recycled paper.

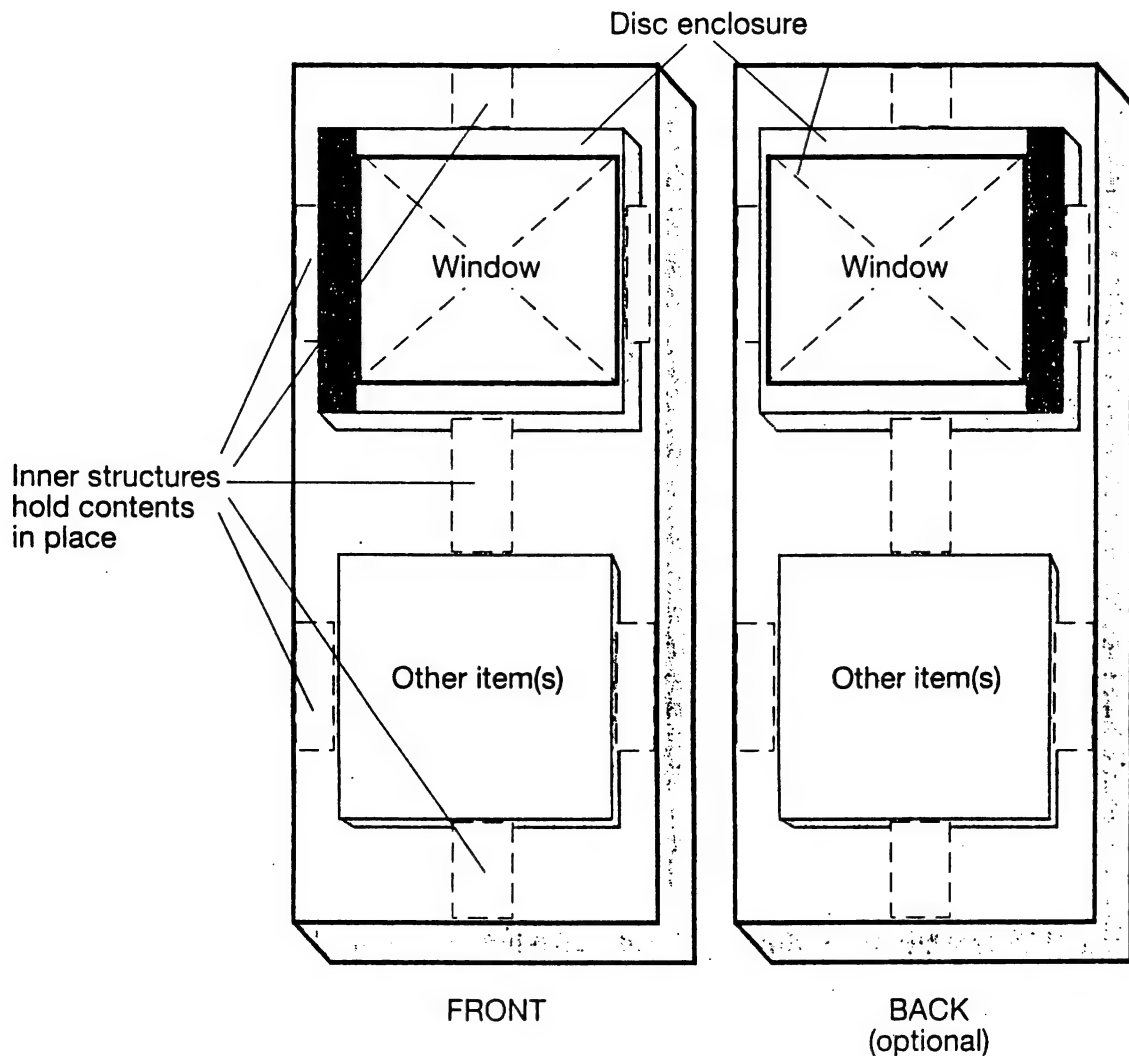
INNER CONSTRUCTION

Each long box must be specially die cut to provide for the proper inner construction which provides both support and strength for the box itself and holds the contents in place.

The positioning of the disc package within the window long box must be precise, so that the the correct view of the package is provided.

On the Master Guide, a graphic indicates the correct positioning of the disc package within the window long box.

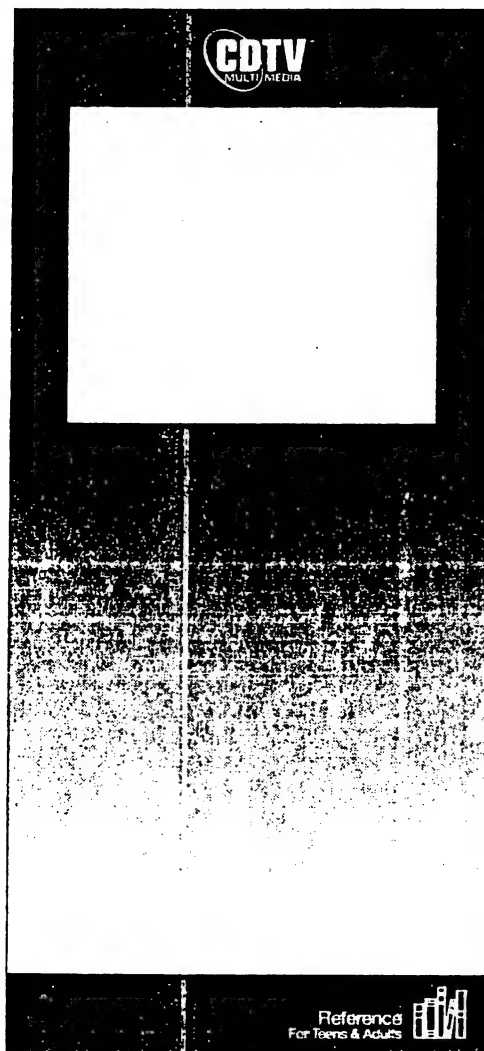
Should you choose to include additional materials, such as reference guides, "smart cards", etc., you need to provide a proper inner structure to the long box so that they are held in place.



GRAPHIC DESIGN

With this manual you received a Window Long Box MASTER GUIDE, which is a large, acetate template. This template shows an actual-size diagram of an unconstructed window long box and complete instructions for creating your artwork.

Below is a depiction of the final result of those instructions. There areas provided on the side and back panels for you to place your own graphics depicting the application title and your company's logo and information.



FRONT

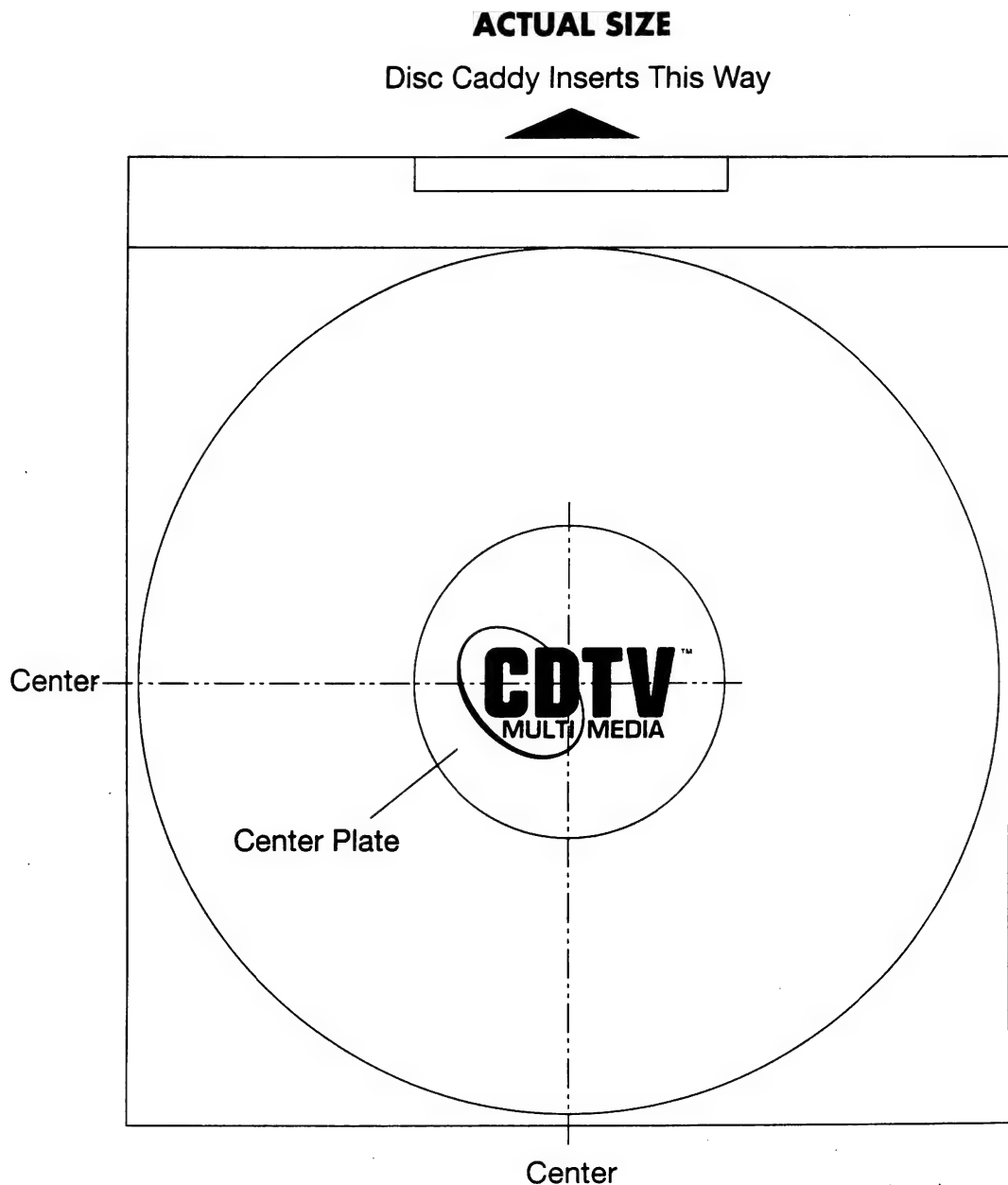
For all instructions regarding the graphic design of the window long box, refer to the film template labelled CDTV WINDOW LONG BOX: MASTER GUIDE provided with this manual.

Section 4: DISC CADDY

When a disc caddy is included in your package, it is recommended that a CDTV MULTIMEDIA logo be imprinted on the disc-shaped plate found in the center of the lid.

Use the following diagram for size and placement only. **DO NOT REPRODUCE FROM THIS DIAGRAM.** For reproduction proofs, see Section 10.

CDTV LOGO PLACEMENT



DO NOT REPRODUCE FROM THIS DIAGRAM

For reproduction proofs, see Section 10

Section 5: CADDY SLEEVE

This chapter will help you prepare the sleeve that encloses a disc caddy.

It is important to bear in mind that the caddy sleeve serves two vital functions:

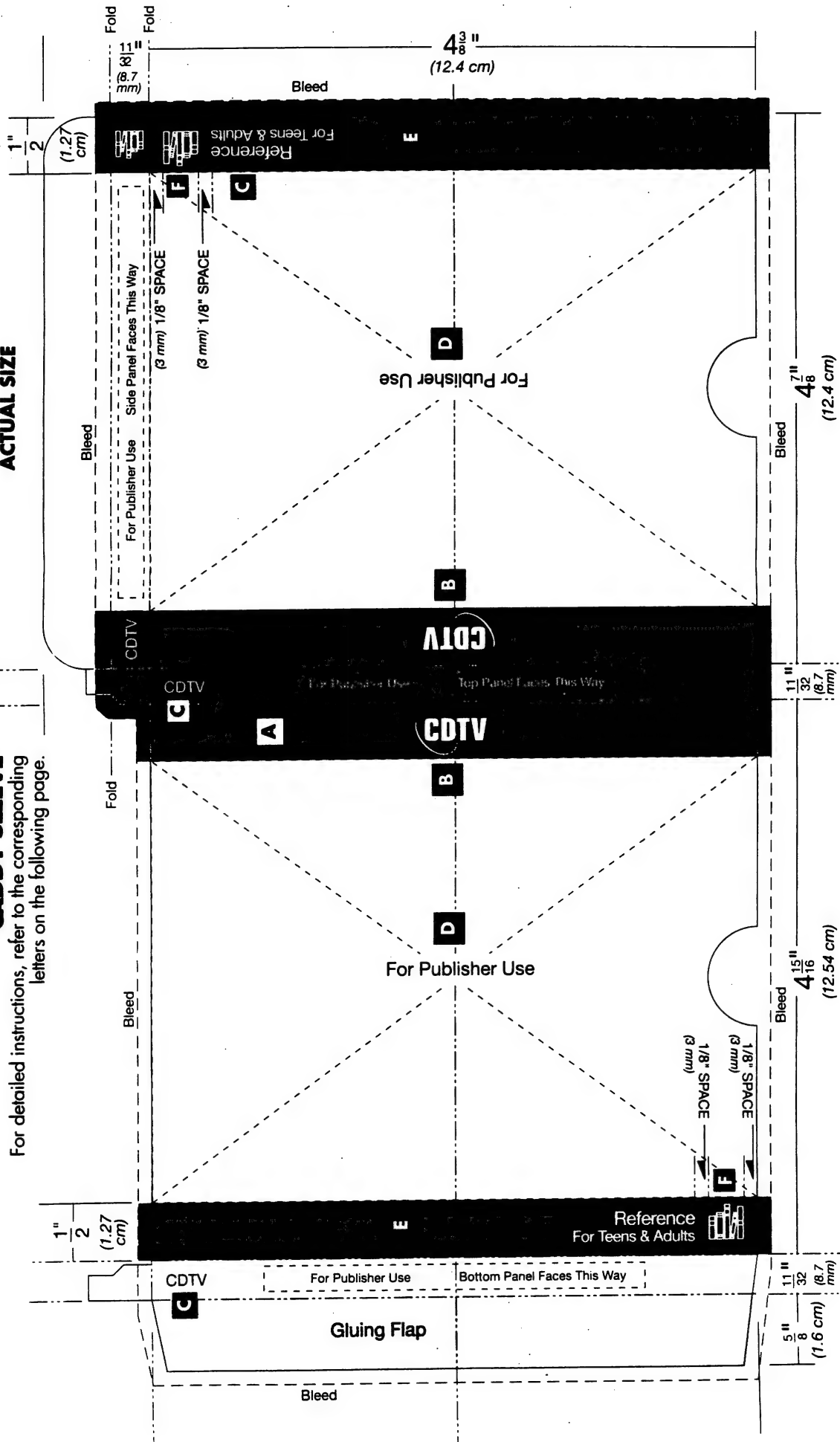
1. **After purchase, the caddy sleeve becomes the primary graphic associated with the disc by the consumer.**
2. **When used inside a window long box, the caddy sleeve becomes the primary retail display graphics.**

For these reasons and others, it is important that the caddy sleeve be prepared properly.

On the following page, you will find a diagram of the dimensions and layout of your caddy sleeve. The diagram is provided exactly to size. Logos, graphics and type are provided for size & position only. **DO NOT REPRODUCE THEM FROM THIS DIAGRAM.** For reproduction proofs, refer to Section 10.

CADDY SLEEVE
For detailed instructions, refer to the corresponding letters on the following page.

ACTUAL SIZE



DO NOT REPRODUCE FROM THIS DIAGRAM
For reproduction proofs, see Section 9.

Your design should consist of the following elements (The letters correspond to the diagram on the preceding page. Follow these letters for detailed instructions for each element of the design):

A TOP BARS: A band of 100% black, .5" tall along the top edge of the front and back covers. The top panel should also print 100% black.

B CDTV MULTIMEDIA LOGO: Centered within both top bars in REVERSE form. Follow examples for size & position. (See Section 10 for reproduction proofs.)

C TYPOGRAPHY:

CATEGORY: Your title category set in **12 pt. Helvetica Regular, upper & lower case. Reverse white* from color bar. Place as shown.** (See page g to find which category applies to your title.)

AGE GROUP: Set in **10 pt. Helvetica Regular, upper & lower case. Reverse white* from color bar. Place as shown.** (See page g for correct age group phrasing.)

CDTV: The letters CDTV set in **10 pt. Helvetica Regular, all caps. Reverse white* from top panels & color bars; surprint 100% black on bottom panel.**

D FOR PUBLISHER USE: The areas within these dashed borders are free for you to use for title graphics and publisher information.

E COLOR BARS: .5" bands of color at bottom edge of front and back panels identifying the title category.

<i>Category</i>	<i>Pantone Ink</i>	<i>Process Equivalent</i>
Arts & Leisure	PMS 299	20%M, 100%C
Education	PMS 108	100%Y
Entertainment	PMS 032	100%Y, 100%M
Music	PMS 265	70%M, 60%C
Periodicals	PMS 354	100%Y, 100%C
Productivity	PMS 021	90%Y, 60%M
Reference	PMS 160	100%Y, 60%M, 40%K

F CATEGORY SYMBOL: (See Section 10 for reproduction proofs.)

Symbol reverses white* from color bars.

***NOTE:** (EXCEPT for Education category, which surprints 100% black).

Section 6: COVER INSERT

Should you choose to enclose your disc in a jewel case, this chapter will help you prepare the insert that will be placed inside the jewel case's transparent cover.

PHYSICAL FORMAT

The cover insert may take the form of a single sheet of paper or a multi-page booklet that contains instructional or supplemental information.

However, for the purpose of these standards, we are concerned only with the visible portion of the insert as it appears in the jewel case, or as we refer to it, the COVER INSERT. If your insert is a multi-page booklet, then the COVER is the only portion to which these instructions apply.

It is important to bear in mind that the cover insert serves two vital functions:

1. **The cover insert, when in place, becomes the cover of the jewel case and, as a result, the primary graphic associated with the disc by the consumer.**
2. **When used inside a window long box, the cover insert becomes the primary retail display graphics.**

For these reasons and others, it is important that the cover insert be prepared properly.

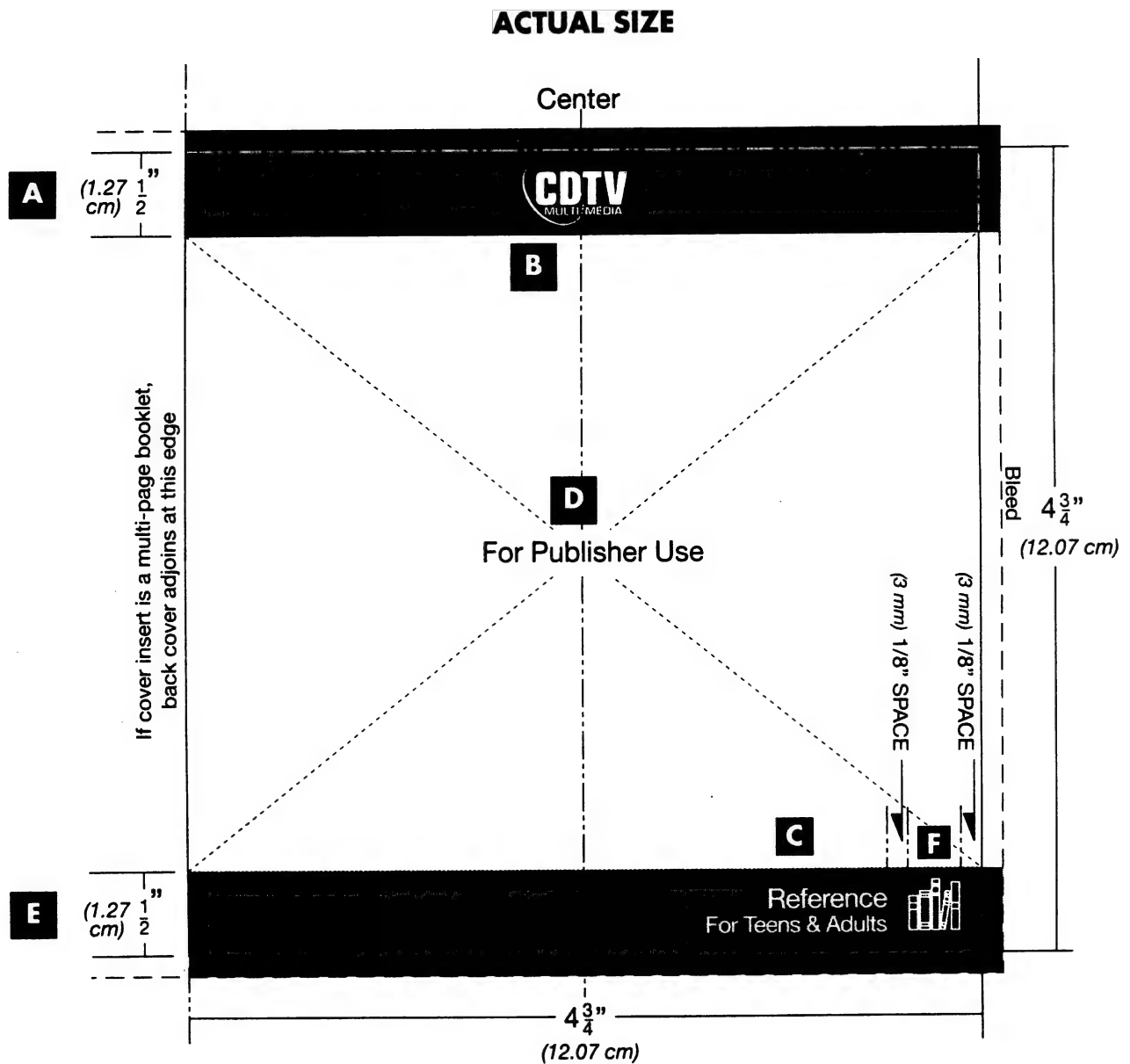
On the following page, you will find a diagram of the dimensions and layout of your booklet cover. The diagram is provided exactly to size. Logos, graphics and type are provided for size & position only. **DO NOT REPRODUCE THEM FROM THIS DIAGRAM.** For reproduction proofs, refer to Section 10.

MATERIALS

Before you choose the paper stock on which you print your cover insert, check with your disc duplicator or fulfillment house for their specifications or requirements. The wrong choice could result in jamming or damage during the insertion process.

PHYSICAL FORMAT & GRAPHIC DESIGN

For detailed instructions, refer to the corresponding letters on the following page.



DO NOT REPRODUCE FROM THIS DIAGRAM
For reproduction proofs, see Section 10

Your design should consist of the following elements (The letters correspond to the diagram on the preceding page. Follow these letters for detailed instructions for each element of the design):

- A TOP BAR:** A band of 100% black, .5" tall, bleeds off the top and right edges of your cover insert. (Bleeds three sides if single sheet insert.)
- B CDTV MULTIMEDIA LOGO:** Centered within the top bar in REVERSE form. Follow example for size & position. (See Section 10 for reproduction proofs.)

C TYPOGRAPHY:

CATEGORY: Your title category set in **12 pt. Helvetica Regular, upper & lower case. Reverse white* from color bar. Place as shown.** (See page g to find which category applies to your title.)

AGE GROUP: Set in **10 pt. Helvetica Regular, upper & lower case. Reverse white* from color bar. Place as shown.** (See page g for correct age group phrasing.)

- D FOR PUBLISHER USE:** The areas within these dashed borders are free for you to use for title graphics and publisher information.

E COLOR BAR:

.5" band of color that identifies the title category. Bleeds off the bottom and right edges of your cover insert. (Bleeds three sides if single sheet insert.)

<i>Category</i>	<i>Pantone Ink</i>	<i>Process Equivalent</i>
Arts & Leisure	PMS 299	20%M, 100%C
Education	PMS 108	100%Y
Entertainment	PMS 032	100%Y, 100%M
Music	PMS 265	70%M, 60%C
Periodicals	PMS 354	100%Y, 100%C
Productivity	PMS 021	90%Y, 60%M
Reference	PMS 160	100%Y, 60%M, 40%K

- F CATEGORY SYMBOL:** (See Section 10 for reproduction proofs.)

Symbol reverses white* from color bar.

***NOTE:** (EXCEPT for Education category, which surprints 100% black).

Section 7: INLAY CARD

When your disc is packaged inside a jewel case, you will need to prepare an "inlay card", which is formed into the case itself, becoming the graphics for the back and side panels.

PHSYCIAL FORMAT

The inlay card is simply a flat sheet of paper that is folded approx. 1/4" from both the left and right edges. This results in one large panel flanked by two narrow panels. When encased inside the jewel case, the large panel becomes the back cover and the narrow panels become the side panels, or "spines".

It is important to bear in mind that the inlay card serves two vital functions:

- 1. The inlay card becomes the back cover and both side panels of the jewel case, and as a result, is viewed prominently by the consumer.**
- 2. When used inside a window long box, the inlay card becomes visible at retail.**

For these reasons and others, it is important that the inlay card be prepared precisely.

On the following page, you will find a diagram of the dimensions and layout of your inlay card. The diagram is provided exactly to size. Logos, graphics and type are provided for size & position only. **DO NOT REPRODUCE THEM FROM THIS DIAGRAM.** For reproduction proofs, refer to Section 10.

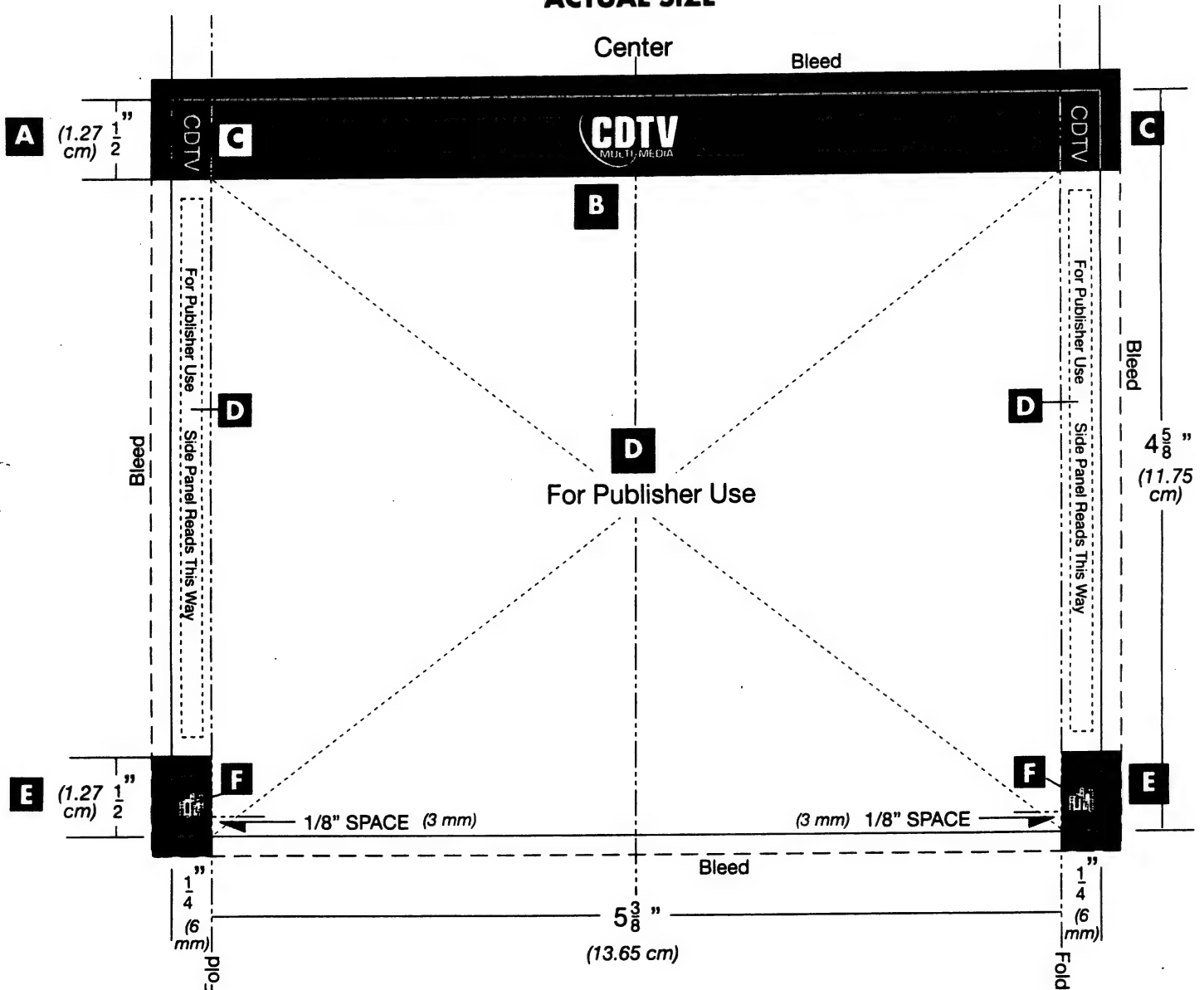
MATERIALS

Before you choose the paper stock on which you print your inlay card, check with your disc duplicator or fulfillment house for their specifications or requirements. The wrong choice could result in jamming or damage during the insertion process.

PHYSICAL FORMAT & GRAPHIC DESIGN

For detailed instructions, refer to the corresponding letters on the following page.

ACTUAL SIZE



DO NOT REPRODUCE FROM THIS DIAGRAM
For reproduction proofs, see Section 10

INLAY CARD

continued

Your design should consist of the following elements (The letters correspond to the diagram on the preceding page. Follow these letters for detailed instructions for each element of the design):

A TOP BAR: A band of 100% black, .5" tall bleeds off the top and both sides of the inlay card.

B CDTV MULTIMEDIA LOGO: Centered within the top bar in REVERSE form. Follow example for size & position. (See Section 10 for reproduction proofs.)

C TYPOGRAPHY:

CDTV: The letters CDTV set in **10 pt. Helvetica Regular, all caps. Reverse white*** from both side panels. Place as shown.

D FOR PUBLISHER USE: The areas within these dashed borders are free for you to use for title graphics and publisher information.

E COLOR BAR:

.5" band of color on both side panels identifying the category. Bleeds off the bottom and outside edge of the inlay card.

<i>Category</i>	<i>Pantone Ink</i>	<i>Process Equivalent</i>
Arts & Leisure	PMS 299	20%M, 100%C
Education	PMS 108	100%Y
Entertainment	PMS 032	100%Y, 100%M
Music	PMS 265	70%M, 60%C
Periodicals	PMS 354	100%Y, 100%C
Productivity	PMS 021	90%Y, 60%M
Reference	PMS 160	100%Y, 60%M, 40%K

F CATEGORY SYMBOL: (See Section 10 for reproduction proofs.)

Symbol reverses white* from color bar on both side panels.

***NOTE:** (EXCEPT for Education category, which surprints 100% black).

Section 8: DISC GRAPHICS

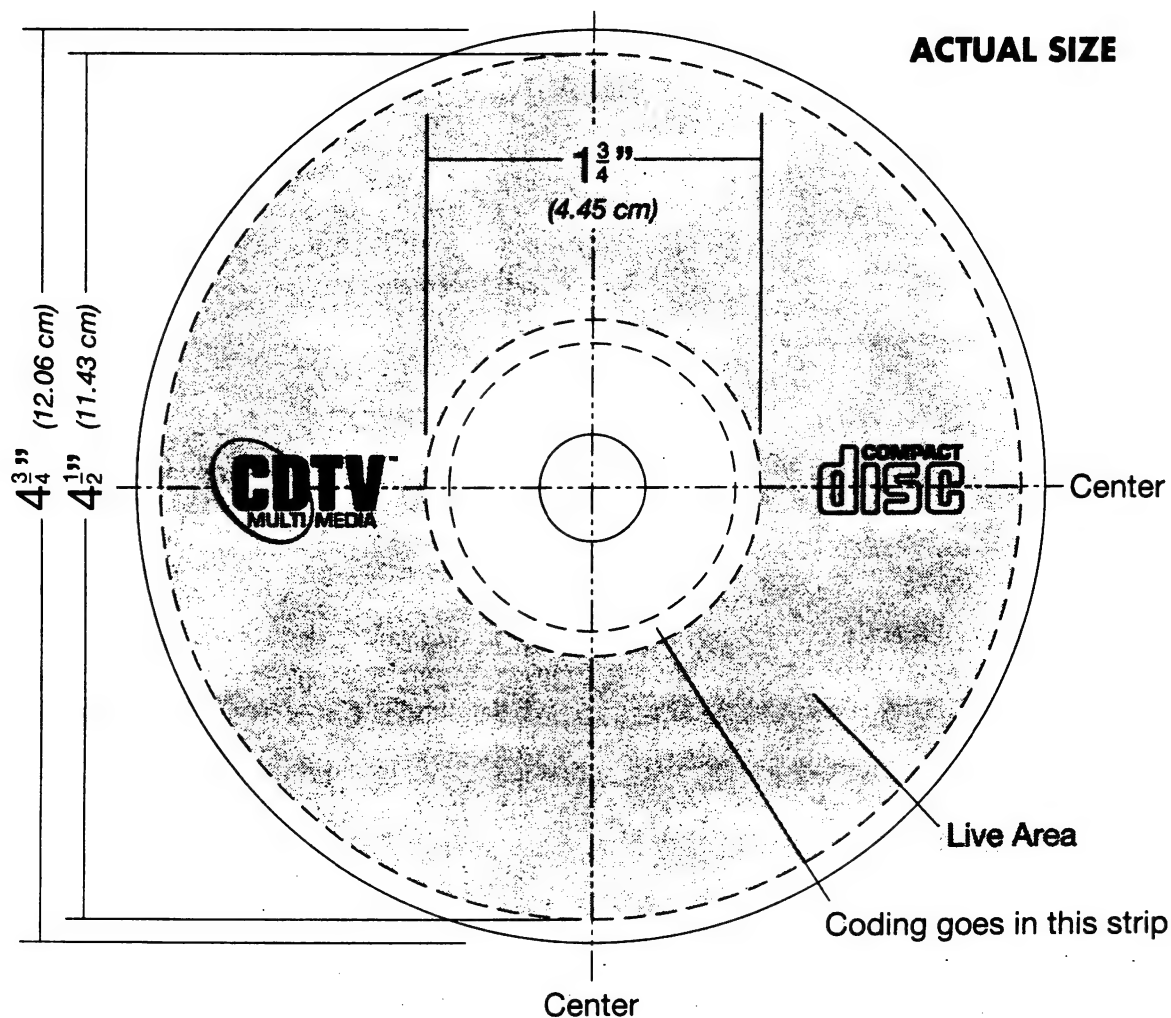
This section will demonstrate the proper application of graphics on the face of the disc itself. Our only specifications are the consistent application of the CDTV MULTIMEDIA logo, and the use of the Compact Disc Logo, which is a requirement of international data standards.

GRAPHIC DESIGN

Below is a diagram of the disc and its design. The diagram is provided exactly to size. The graphic on the disc are provided for size & position only. **DO NOT REPRODUCE FROM THIS DIAGRAM.** For reproduction proofs, refer to Section 10.

CDTV MULTIMEDIA LOGO: Printed in positive form, positioned squarely at "9 o'clock" on the disc, square to all other graphics. (See Section 10 for reproduction proofs.)

COMPACT DISC LOGO: Printed in positive form, positioned squarely at "3 o'clock" on the disc, square to all other graphics. (See Section 10 for reproduction proofs.)

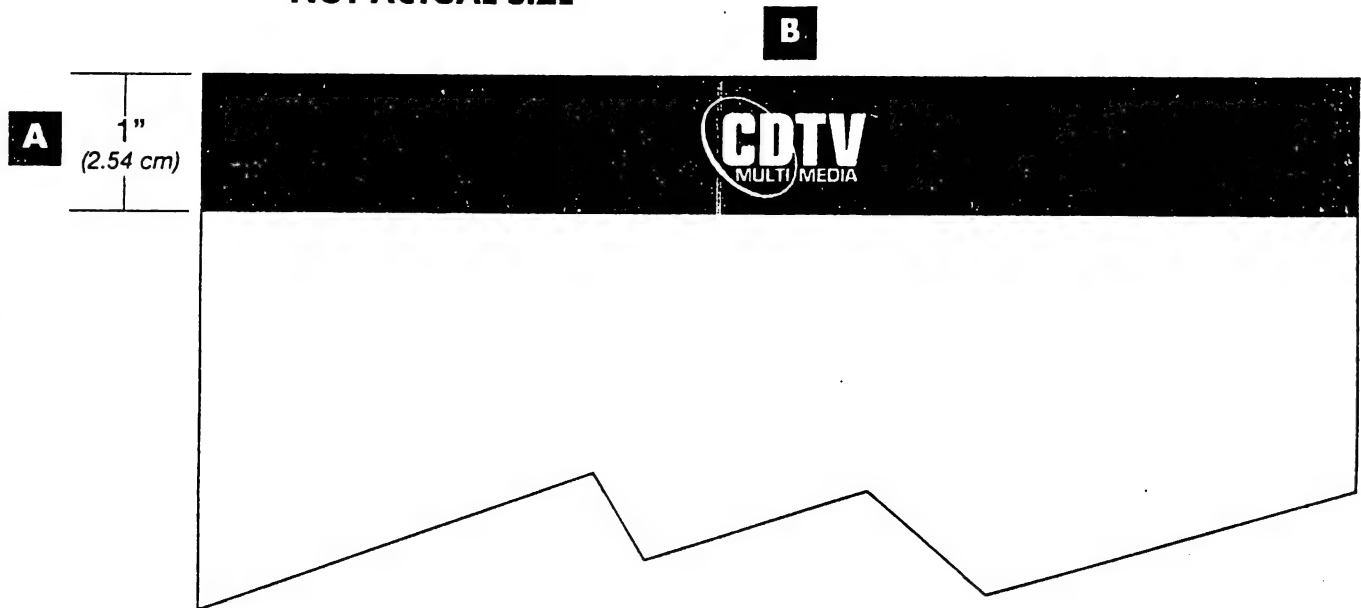


Section 9: PROMOTIONAL MATERIALS

In this section, you will find recommendations for the use of the CDTV MULTIMEDIA logo on promotional materials such as sell sheets or, as they are also known, "slicks".

We have depicted below a recommended graphic for the head of an 8-1/2 x 11" sheet. As we all know, promotional materials can assume a limitless number of sizes and formats. However, we ask that you adapt this guideline to your format, scaling its dimensions proportionately.

NOT ACTUAL SIZE



A TOP BAR: A band of 100% black, 1" tall bleeds off the top and both sides of the sheet.

B CDTV MULTIMEDIA LOGO: Centered within the top bar in REVERSE form. Follow example for size & position. (See Section 10 for reproduction proofs.)

Section 10: REPRODUCTION PROOFS

In this section, you will find reproduceable artwork, or "stats" of the graphics you will need to follow the directions in this manual.

ALWAYS USE PHOTOMECHANICAL REPRODUCTIONS OF THE REPRODUCTION PROOFS.

NEVER...

- **Use photocopies in your artwork.**
- **Attempt to re-create the artwork yourself.**
- **Reproduce art from the diagrams or templates.**

FOR COMPUTER GRAPHICS USERS:

Digitized versions of all of the following logos and symbols are available for those who employ computerized systems to create their artwork.

For further information, contact:

Communiqué
30 Galesi Drive
Wayne, New Jersey 07470
(201) 785-9115
Art Directors: Jeff Jackson & Bob Debiak

CDTV MULTIMEDIA & COMPACT DISC LOGO
Reproduction Proofs



For use on Long Box/Window Long Box/Disc Caddy



For use on Promotional Materials













































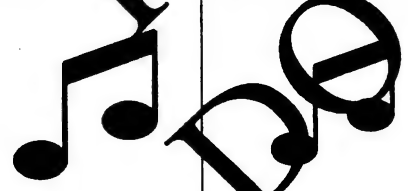






















































For use on Disc Graphics



For use on Caddy Sleeve/Cover Insert/Inlay Card



CATEGORY SYMBOLS: Reproduction Proofs

For use on...	Long Box/ Window Long Box	Cover Insert/ Caddy Sleeve (Side Panel)	Caddy Sleeve (Front/Back Panels)	Inlay Card
Arts & Leisure 		   	   	   
Education 		   	   	   
Entertainment 		   	   	   
Music 		   	   	   
Periodicals 		   	   	   
Productivity 		   	   	   
Reference 		   	   	   

GLOSSARY

Bleed	Extending graphics beyond the edge as a safety margin.
Booklet	Pamphlet that accompanies discs. It can be inserted into the clear cover of a jewel case, thus becoming the cover of the case itself.
Disc Caddy	Cartridge into which discs must be placed before insertion into CDTV player. It provides the stability necessary for correct data transfer from the disc.
Caddy Sleeve	Paperboard box or wrapping that encloses a disc caddy.
Category	CDTV titles of a similar topic or focus.
Control space	A margin of empty space provided around a logo or graphic to ensure that no other elements "crowd" it.
Inlay Card	Printed sheet that becomes encased inside the back wall of a jewel box, thus visibly becoming the back and side panels of the case itself.
Jewel Case	Standard plastic container for compact discs with hinged, transparent lid.
Live area	Safe area in which graphics can be applied.
Logo	The official symbol of CDTV products.
Long Box	Outer package used conventionally for compact discs, measuring approx. 12.3" tall, 5.7" wide, .5" deep when constructed.
Publisher	Any organization that creates and markets CDTV titles.
Pantone/PMS	Universally accepted standard for color matching in the printing industries that uses a numbered assortment of colors.
Process	The printing method of simulating vast ranges of color by using combined patterns of minute dots of 4 standard color inks – yellow, magenta, cyan & black (known as the process colors).
Reproduction proofs	Artwork that is provided in a clean, perfect form in order to ensure optimum quality in reproduction.
Window Long Box	Modification of standard long box with die cut opening on the front and back panels allowing a view of the disc package inside.

Modification for Switchable PAL/NTSC CDTV

Background

CDTV units are capable of generating both PAL and NTSC video formats. Older CDTV units were configured with jumpers at the factory for one or the other format. The jumpers were used instead of a switch to insure FCC compliance. Certain developers or demonstrations may require that their older CDTV unit be easily switchable between formats to test or run applications designed for different world markets. For these units, it is possible to replace the jumpers with a switch so that the CDTV unit can be easily changed from one video format to the other.

Tools & Materials

To make this modification you will need the following tools:

1. Soldering iron with fine tip
2. Desoldering device (solder wick, etc.)
3. X-acto knife
4. Wire strippers
5. Needlenose pliers or tweezers

The following materials will be required:

1. Small 3PDT or 4PDT switch
2. Wire-wrap wire

Familiarization

Remove the top cover (6 screws) and locate oscillators X1 and X5 on the left side of the board. These are used for NTSC and PAL signal generation respectively. Jumpers JP8–11 select between PAL and NTSC formats. The jumper configuration and functions are listed in the following table:

<u>Jumper</u>	<u>Format</u>		<u>Description</u>
	<u>NTSC</u>	<u>PAL</u>	
JP8AB	Open	Shorted	PAL crystal output
JP8CD	Shorted	Open	NTSC crystal output
JP9	Shorted	Open	+5V for NTSC crystal
JP10	Open	Shorted	+5V for PAL crystal
JP11	Shorted	Open	Frequency divider select

The NTSC/PAL switch should be located to the left of, and centered between, X1 and X5. If needed, the metal shield to the left of X1 and X5 may be cut to accommodate the switch. The wires, particularly the ones attached to JP8, should be kept as short as possible.

Instructions

1. Remove any soldered connections between the pads of JP8-11. Some units were pre-configured to NTSC by connecting in etch the pads of JP8CD, JP9, and JP11. If these pads are connected, cut the traces connecting the pads using the X-acto knife.
2. Make the following connections using the wire-wrap wire. Note that the individual (SW)itch elements are listed as A, B, C, and D and the switch terminals are listed as (C)ommon, (1)st position and (2)nd position. Also note that the individual pads of JP8-11 are referenced on the included schematics and silkscreen drawings.

1___	SWA pin C to JP9 pad A	6___	SWC pin C to JP8 pad B or D
2___	SWA pin 1 to JP9 pad B	7___	SWC pin 1 to JP8 pad C
3___	SWA pin 2 to JP10 pad B	8___	SWC pin 2 to JP8 pad A
4___	SWB pin C to JP11 pad B	*SWD not used	
5___	SWB pin 1 to JP11 pad A		

3. Attach the switch to the circuit board with some type of non-conductive adhesive (a hot melt glue gun works well for this).

Testing and Operation

1. Position 1 of the PAL/NTSC switch selects NTSC format, position 2 selects PAL format. The format must be selected before turning on the main power switch of the CDTV unit.
2. To test the switch operation, connect a multisync monitor to the CDTV unit. Select NTSC format and turn on the main power. The CDTV power up logo should fill the entire screen of the monitor.
3. Turn off the CDTV, select PAL format, and turn the power on again. The logo should be smaller than the one displayed in NTSC format.
4. After verifying the correct operation of the PAL/NTSC switch, turn off the power and replace the cover.

International Keyboard Input

The Amiga computers are sold internationally with a variety of local keyboards which match the standards of particular countries.

The V2.0 *KeyShow* (or V1.3 *KeyToy*) command shows you a graphical representation of the keyboard for the currently installed keymap. If you would like to see any of these national keyboard layouts, you can use the *SetMap* command in a shell to set that shell to any keymap from *devs/keymaps*, and then call *KeyShow* in that same shell to see a display of the keyboard that corresponds to the keymap.

If you look at several, you will see that some letters and special symbols are in different physical positions on the various keyboards. For instance, on the German and Italian keyboards, the Y and Z keys are swapped when compared to the USA keyboard. Since the physical position of a key determines the raw key code that it generates, straight RAWKEY input is not internationally compatible. Pressing the second key on the fifth row will generate the same raw key code on all Amiga keyboards, but will be decoded as a Z on a US keyboard and as a Y on a German.

Since V1.2, the Amiga *console.device* supports national keyboards by providing mapping of raw key codes to the proper ASCII characters and strings as specified in a Keymap. In addition, the *console.device* provides handling of the "dead keys" used to generate accented characters. Any keyboard input processed by the *console.device* will be automatically translated to the installed keymap.

All of the national Keymaps, including *usa*, contain dead keys. Basically, a dead key is a key that produces no output until a second key is pressed. Thus the dead key modifies the output of another key. Often, a dead key is used to indicate that a particular accent mark is to be placed on the next character typed. The ASCII values for these characters are all greater than 127. They are listed in the International Character Codes table in the AmigaDOS manual. If a dead key is used to request an invalid accent for a character, the normal unaccented character results.

On the USA keyboard, the F, G, H, J, and K keys are defined as dead keys. Pressing Alt and one of these keys is a request for an accent to be placed on the next character you type.

For example, type these in a shell or editor with the default *usa* keymap:

ALT-F	then A results in an A accented with ' <i>Á</i>
ALT-H	then E results in an E accented with ^ <i>Ê</i>
ALT-J	then C results in a plain C (invalid accent for C)

KeyShow Shows Alt Accents. If you press the Alt key while using *KeyShow*, the keycaps of the dead keys (for example, the F, G, H, J and K keys with the *usa* Keymap) will display the accents generated by those keys.

Since V1.2, the *console.device*, IDCMP VANILLAKEY, and AmigaDOS CON: and RAW: all provide automatic handling of dead keys and translation of raw keycodes to ASCII based on the current keymap. If your software requires non-VANILLA keys such as the cursor and function keys, using *console.device* keyboard input in your Intuition window will allow you to receive the escape sequences generated by these keys. If you are using IDCMP RAWKEY input in international software, you must use the *console.device*'s *RawKeyConvert()* function properly to get keymap

translation and dead key handling. Use NULL for the keyMap argument to get translation to the currently installed Keymap.

Do The Proper Setup. You must `OpenDevice()` the *console.device* and set up a `ConsoleDevice` base variable from `io_Device` before using `RawKeyConvert()`. For an example, see the `DeadKeyConvert()` routine in the "Intuition" chapter example *MouseKeys.c* of the V1.3 *Amiga ROM Kernel Reference Manual: Libraries and Devices*. Also note that in V2.0, you can ask for `RAWKEY|VANILLAKEY` together to get keymap translated ASCII `VANILLAKEY` messages for the alphanumeric keys and raw `RAWKEY` values for the special keys (function keys, help key, etc.).

Be sure to test your code with ASCII characters greater than 127 such as accented characters (e.g., Alt-F A) wherever your code accepts keyboard input to ensure that your logic and data structures work properly with high ASCII values.

Make sure that your case-insensitive string compare is internationally compatible and properly handles the equality of shift and unshifted accented characters. The V2.0 *utility.library* provides two internationally compatible string compare functions, `Stricmp()` and `Strnicmp()`.

Test your code after using *SetMap* to install various keymaps, and try out all of your text entry, filename entry, and other keyboard features with strings containing accented characters wherever applicable. Use *KeyShow/KeyToy* as a guide for what the keycaps show in each country and make sure your program is receiving the proper characters for that keymap. You won't be able to test all of the keys of some national keyboards, but you can test enough keys to be certain your program is getting proper ASCII translation.

Finding the Aspect Ratio

The pixel aspect ratio describes the ratio of the width (xAspect) to height (yAspect) of the pixels in a Screen or ViewPort. In order to create a truly What-You-See-Is-What-You-Get (WYSIWYG) graphics display on the Amiga, you need to find the pixel aspect ratio of the display mode you are using. With the proper aspect ratio, an application can correctly display and store ILBM files, can rotate objects properly, and can calculate the proper dimensions to draw true circles and squares so they appear the same on the Amiga display as they would on some other output device like a laser printer.

Under V1.3 and the original Amiga chip set, relatively few display modes were available. Under V2.0 versions of the OS, applications can use hard-coded values for the X/Y pixel aspect ratio. The aspect ratios for the display modes available to the 1.3 system are (xAspect / yAspect):

NTSC Lores 44/52

PAL Lores 44/44

Halve the xAspect for Hires modes. Halve the yAspect for Interlaced modes.

These aspect values are more accurate than the values in the original IFF document.

On PAL displays, the pixels of Lores screens and Hires-Interlaced screens are square, as they have an aspect ratio of 44/44 and 22/22, respectively. To draw a square 100 pixels wide in one of these PAL modes, you could simply draw a square that is 100 pixels x 100 pixels. On a PAL Lores-Interlace display, the Y resolution is doubled, making each pixel half as tall, so you would have to draw a rectangle that was 100 pixels x 200 pixels to get the same size square.

On an NTSC display, pixels are not square. Pixels on a Lores NTSC screen have an aspect ratio of 44/52 (or 11/13). This means that each pixel is slightly narrower than it is high.

To draw a true square that is 100 pixels wide in an arbitrary display mode, it is necessary to calculate the correct height for the square based on the pixel aspect ratio.

$$\text{width} / \text{yAspect} = \text{height} / \text{xAspect}$$

(in words, width is to yAspect as height is to xAspect)

If the X/Y aspect is 44/52 (Lores NTSC), the calculation would be:

$$100 / 52 = \text{height} / 44$$

$$\text{height} = (100 * 44) / 52 = 4400 / 52 \text{ /* solve for height */}$$

Because this example uses only integer math, the ratio must be rounded to the nearest integer. A fraction a/b (where a and b are integers) rounded to the nearest integer approximately equals:

$$(a + (b/2)) / b$$

apply this to the ratio above:

$$\text{height} = (4400 + (52 >> 1)) / 52 \text{ /* with rounding */}$$

$$\text{height} = 4426 / 52 = 85.115384... \text{ /* Approximate */}$$

$$\text{height} = 85 \text{ /* truncated */}$$

Therefore, to be square on a 44/52 aspect screen, a 100 pixel wide square would have to be 85 pixels tall.

Under V2.0 and the ECS chip set, the Amiga display is more dynamic. It has many new display modes, each of which has its own distinct pixel aspect ratio. For this reason, it is not practical nor desirable to hardcode the aspect ratios for the modes that you know about (except when running under V1.3). When running under V2.0, the pixel aspect should be determined by querying the display database (see the article "An Introduction to V36 Screens and Windows" from the September/October 1990 issue of Amiga Mail for more information on how to query the display database). A valid **DisplayInfo** structure contains the X and Y aspect in the **Resolution.x** and **Resolution.y** fields. When writing an ILBM under V2.0, use these pixel aspects from the display database for the BMHD chunk's **xAspect** and **yAspect** values.

The following example demonstrates how to determine the X and Y aspects under both V2.0 and V1.3.

```
/* getaspect.c - Execute me to compile me with SAS C 5.10
LC -bl -cfistq -v -y -j73 getaspect.c
Blink FROM LIB:c.o,getaspect.o TO getaspect LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
Gets X/Y pixel aspect of a screen's ViewPort
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <graphics/displayinfo.h>
#include <graphics/gfxbase.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); } /* really */
#endif

#define MINARGS 1

UBYTE *vers = "\0$VER: getaspect 37.1";
UBYTE *Copyright =
    "getaspect v37.1\nCopyright (c) 1990 Commodore-Amiga, Inc. All Rights Reserved";
UBYTE *usage = "Usage: getaspect";

void bye(UBYTE *s, int e);
void cleanup(void);

struct Library *IntuitionBase;
struct Library *GfxBase;

void main(int argc, char **argv)
{
    struct Screen *first;
    struct ViewPort *vp;
    struct DisplayInfo DI;
    ULONG modeid;
    UBYTE xAspect, yAspect;

    if (((argc)&&(argc<MINARGS)) || (argv[argc-1][0]!='?'))
    {
        printf("%s\n%s\n", Copyright, usage);
        bye("", RETURN_OK);
    }

    /* We will check later to see if we can call V36 functions */
    IntuitionBase = OpenLibrary("intuition.library", 34);
```

```

GfxBase = OpenLibrary("graphics.library",34);

if ((!IntuitionBase) || (!GfxBase))
    bye("Can't open intuition or graphics library",RETURN_FAIL);

printf("Using front screen's ViewPort (for example purposes only):\n");

first = ((struct IntuitionBase *)IntuitionBase)->FirstScreen;
vp = &first->ViewPort;

xAspect = 0;    /* So we can tell when we've got it */

if (GfxBase->lib_Version >= 36)
{
    modeid = GetVPMODEID(vp);
    if (GetDisplayInfoData(NULL, (UBYTE *)&DI, sizeof(struct DisplayInfo),
        DTAG_DISP, modeid))
    {
        printf("Running 2.0, ViewPort modeid is %08lx\n",modeid);
        xAspect = DI.Resolution.x;
        yAspect = DI.Resolution.y;
        printf("Pixel xAspect=%ld yAspect=%ld\n",xAspect, yAspect);
        printf("PaletteRange is %ld\n",DI.PaletteRange);
    }
}

if (!xAspect) /* pre-2.0 or GetDisplayInfoData() failed */
{
    modeid = vp->Modes;
    printf("Not running 2.0, ViewPort mode is %04lx\n",modeid);
    xAspect = 44; /* default lores pixel ratio */
    yAspect = ((struct GfxBase *)GfxBase)->DisplayFlags & PAL ? 44 : 52;
    if (modeid & HIRES)
        xAspect = xAspect >> 1;
    if (modeid & LACE)
        yAspect = yAspect >> 1;
    printf("Pixel xAspect=%ld yAspect=%ld\n",xAspect, yAspect);
}

bye("",RETURN_OK);
}

void bye(UBYTE *s, int e)
{
    cleanup();
    exit(e);
}

void cleanup()
{
    if (GfxBase)
        CloseLibrary(GfxBase);
    if (IntuitionBase)
        CloseLibrary(IntuitionBase);
}

```

CDTV Technical Specifications

Video Outputs

- Analog RGB, Digital RGBI (DB-23 connector)
- Composite video NTSC (RCA connector)
- Component video Y-C (S connector type for S-VHS and Hi8)
- RF Modulated (F connector)
- Optional genlock capabilities via plug-in module:
 - three-mode (CDTV, video source or mixed) under software control

Video Display

- 400 lines/Vertical frequency 60Hz(NTSC); 512 lines/Vertical frequency 50Hz(PAL)
- Graphic coprocessor with beam synced draw, fill, and move modes (blitter)
- Maximum 1MB video memory (chip memory)
- Palette of 4096 colors
- 8 sprites per scanline

Microprocessor

- Motorola 68000 16/32 bit 7.16 MHz (NTSC); 7.09 MHz (PAL)

Custom Chips

- Three Amiga-specific custom chips (Agnus, Paula and Denise) that enhance system performance by taking over tasks such as handling video, sound, direct memory access (DMA), and/or graphics

Memory

- 1MB Chip RAM
- 2K non-volatile RAM reserved for system
- 512K ROM

Internal Slots

- Intelligent video slot (for optional genlock, RF board, etc.) DMA slot for SCSI, LAN, etc.

CD Audio Specs

8X oversampling	
Audio output	External 1.4 VRMS, 10K OHM
Frequency response	4–20KHz
Signal/Noise	-102db
Channel Separation	-92db
Harmonic Distortion	0.02% at 1KHz
Maximum audio capacity	28 hours—AM quality
Sample Rates	variable from CD audio rate (44KHz) to 6KHz
Dual 16-bit D/A converter	plus 64 levels of attenuation

CD-ROM Drive Specs

Sony/Philips type	CD-ROM standard	Mode 1	Mode 2
Data readout from disk	153 KBytes/sec	(Mode 1)	171 KBytes/sec (Mode 2)
Average access time	0.5 sec		
Maximum access time	0.8 sec		
Soft read error	Less than 10e-9		
Hard read error	Less than 10e-12		
Seek error	Less than 10e-6		
Commands	CD-ROM, CD Audio, CD+G		
MTBF	10,000 P.O.H.		
Standard Supported	ISO-9660		
Data Capacity	540 MB		

Front Port

Stereo Headphone Jack
Port for optional personal RAM/ROM card (256K)

Rear Ports

Centronics parallel interface
RS-232 serial interface
External floppy disk drive interface (Amiga compatible)
Hardwired alternative to IR for keyboard, mouse, joystick, 2 audio output ports (RCA-type plug)
MIDI in and out

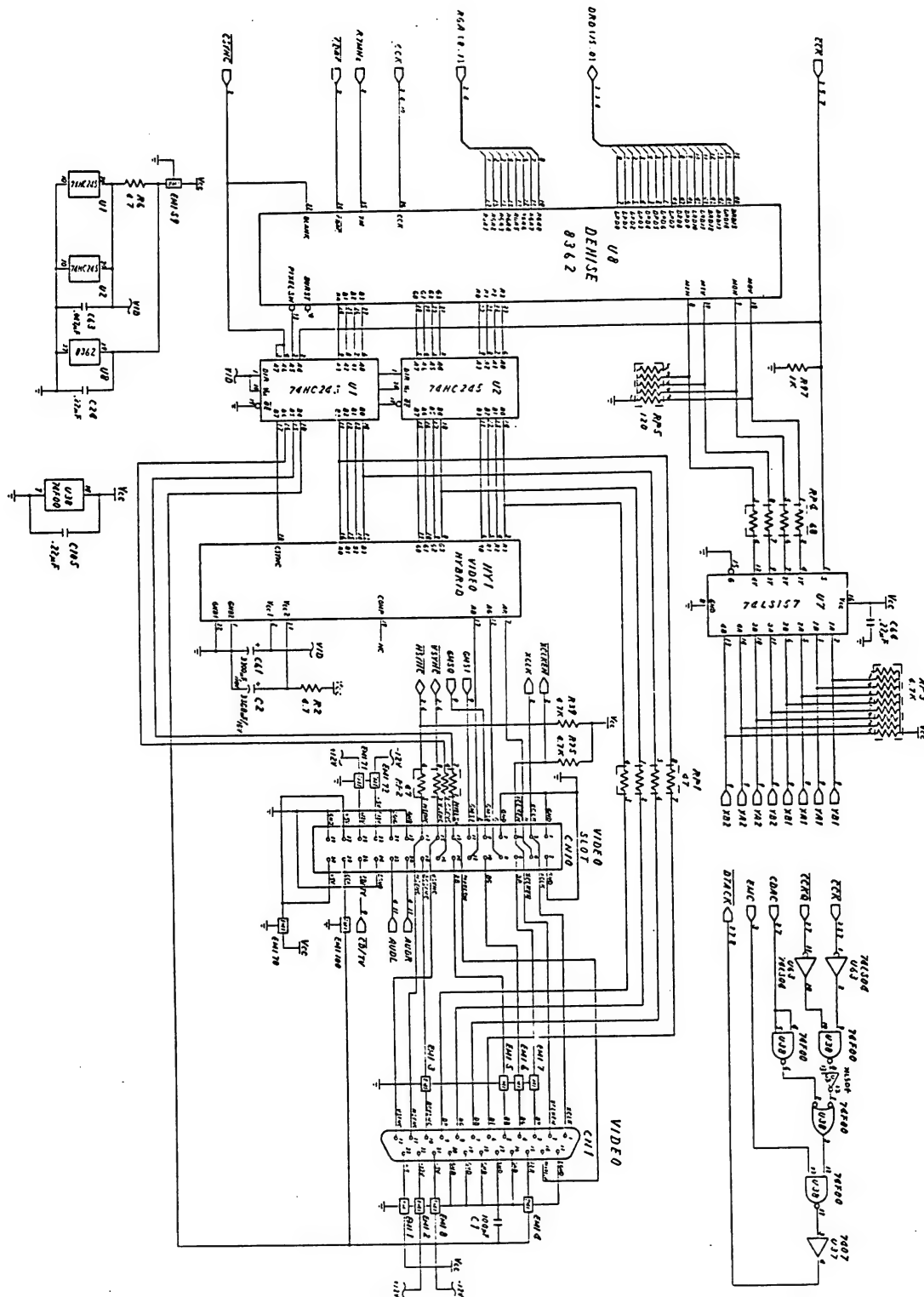
Power Consumption

50W (average AC100-240V, 50/60Hz)

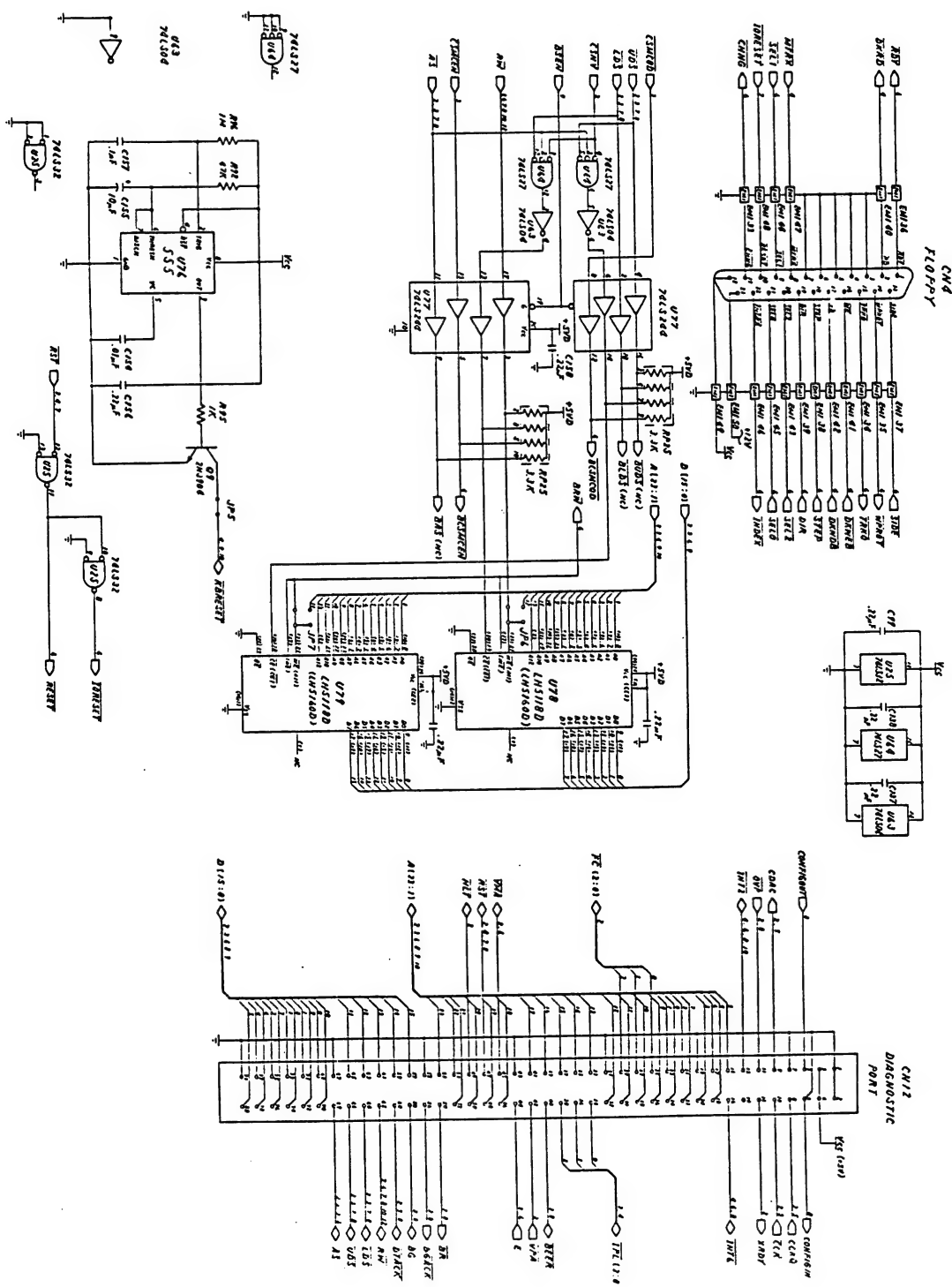
Dimensions

Player	17.25"wx12.5"Dx3.7"H
IR	8.25"Wx2.75Dx.88"H

Video Port Connector



Diagnostic Port Connector



Administrative Forms

**INSTRUCTIONS FOR COMPLETING
CDTV
MANUFACTURING LICENSE**

1. **Make a copy of the License Agreement for your future use.**
2. **Complete and sign two copies of the attached contract.**
3. **Complete and sign a separate copy of the Exhibit for each title you plan to manufacture.**
4. **Return the two signed copies to:**

**CATS Administration
CDTV License Enrollment
1200 Wilson Drive
West Chester, PA 19380**

The license will then be processed and signed by an officer of the corporation and one copy returned to you. You will receive your licensed developer discs and materials shortly after that.

COMMODORE CDTV PRODUCT LICENSE

THIS Agreement is entered into and made effective this _____ day of _____, 199____, by and between Commodore Holding B.V. ("Commodore"), and the LICENSEE identified below.

1.0 DEFINITIONS

1.1 "CDTV Player" means a proprietary system capable of retrieving program information from a CD-ROM, which system is distributed under the trademark "CDTV", and manufactured by Commodore or by a third party under a written license from Commodore.

1.2 "Product" means the work identified in Exhibit 1 (the "Work"), as manufactured hereunder in a CD-ROM format compatible with the CDTV Player, together with such corrections, modifications, improvements and updates to the Product that do not substantially alter its original structure, content or functionality.

1.3 "Licensed Software" means those programs which Commodore makes available hereunder for (a) inclusion by LICENSEE in the Product (the "Application Software"), (b) LICENSEE's use in preparing and testing the Work to be manufactured hereunder (the "Utility Software"), and (c) LICENSEE's use in building a master image of the prepared and tested Work (the "Mastering Software"), together with such corrections, modifications, improvements and updates to the Licensed Software which Commodore makes generally available to licensees of the Licensed Software, without separate charge, and including any instructions, user manuals and other related documentation made available by Commodore to facilitate LICENSEE's use of such Licensed Software.

1.4 "Commodore's Proprietary Rights" means any and all patents, copyrights, and trade secrets, owned or controlled by Commodore during the term of this Agreement, which cover inventions, works and processes used or useful in preparing the Product for manufacture and in making the Product compatible with the CDTV Player.

1.5 "Proprietary Marks" means any and all trade names, trademarks, insignias, logos, proprietary marks, and the like identified in Exhibit 1.

2.0 LICENSE GRANTED

2.1 During the term of this Agreement and subject to the conditions herein, Commodore hereby grants to LICENSEE a nontransferable (except as specified in Paragraph 10.5), nonexclusive, license under Commodore's Proprietary Rights to use the Licensed Software, to manufacture copies of the Product in accordance with this Agreement for worldwide distribution to end-users, as follows:

(a) to reproduce and to modify with Commodore's written approval, the Application Software, and to include the same in the Product,

(b) to employ the Utility Software for the sole purpose of preparing and testing the Work for manufacture hereunder,

(c) to employ the Mastering Software for the sole purpose of building a master image of the prepared and tested Work in a format compatible with the CDTV Player, and

(d) to produce for distribution, CD-ROM copies of such master image as is created pursuant to the immediately preceding paragraphs (a) through (c), it being acknowledged by the parties that such copies may be marketed and distributed to end-users and that end-users may use such Licensed Software as has been included in the Product as a result of the manufacture of such Product hereunder.

2.2 Such license to produce copies of the Product master image as is set forth in Paragraph 2.1 (d), above shall include the right to have a third-party contractor replicate such master image.

2.3 During the term of this Agreement and subject to the conditions herein, Commodore hereby grants to LICENSEE the right to use the Proprietary Marks, only in connection with and as reasonably required for, LICENSEE's manufacture, distribution, and promotion of the Product hereunder, provided that LICENSEE's use of Proprietary Marks shall always be in accordance with Commodore's specifications, policies and directions and shall clearly indicate that the same is the property of Commodore or a Commodore affiliated company, as the case may be, and as reasonably directed by Commodore. LICENSEE acknowledges that Commodore may, in its reasonable judgment, at any time object to a specific use or application of any of the Proprietary Marks, specifying the reason for such objection, in which event LICENSEE will immediately cease such use or application thereof, except that LICENSEE may with Commodore's prior permission, which permission shall not be unreasonably withheld, use a reasonable supply of pre-printed materials that have been ordered or are on hand. All use by LICENSEE of such Proprietary Marks shall inure to the benefit of Commodore and its affiliated companies.

2.4 Except for those rights expressly granted above, no other rights or licenses are granted hereunder, by implication or otherwise. All such other rights, including without limitation the title to and ownership of the Licensed Software (including any modifications thereto), Commodore's Proprietary Rights and Proprietary Marks, as well as all rights relating to the CDTV Player are reserved to Commodore, its licensors and suppliers, and except as aforesaid the title to and ownership of the Product are reserved to LICENSEE, its licensors and suppliers.

3.0 TERM AND TERMINATION

3.1 This Agreement shall continue in full force and effect for a period of two (2) years from the effective date hereof and will be renewable for subsequent two year periods at LICENSEE's option, by written notice to Commodore at least thirty (30) days prior to the termination of any such period, provided that this Agreement shall automatically sooner terminate if LICENSEE fails to publish the Product within a twelve (12) month period following such effective date, or fails to continue to distribute such Product for any period in excess of nine (9) consecutive months, or unless terminated earlier as set forth below.

3.2 In addition to any other remedy which it may have at law or in equity, either party may immediately terminate this Agreement if the other party (a) materially breaches or fails to perform any of its obligations hereunder and if such breach or failure is not capable of being corrected, or if such breach or failure, though correctable, is not corrected within thirty (30) days of notice thereof from the non-breaching party, or (b) becomes insolvent or admits in writing its inability to pay its debts as they mature, or (c) files a voluntary petition in bankruptcy, makes an assignment for the benefit of creditors or has filed against it a petition under any bankruptcy law (and the same is not dismissed within sixty (60) days). Nothing herein shall prevent either party from seeking immediate injunctive relief or other appropriate remedy in the event of the other party's failure to comply with its obligations under Paragraphs 5.1 through 5.5 of this Agreement, the failure to comply with such obligations being deemed not correctable.

3.3 LICENSEE may terminate this Agreement at any time and without cause, upon sixty (60) days written notice to Commodore, providing LICENSEE is not in breach of any of the terms of this Agreement and is current with respect to all payments due hereunder. LICENSEE will remain obligated to pay all accrued amounts that become due and payable to Commodore after any such termination.

4.0 PAYMENTS

4.1 In consideration of the licenses and rights granted hereunder by Commodore, LICENSEE agrees to pay to Commodore the unit license fee set forth in Exhibit 1 ("License Fee"), for each unit of Product that is shipped to its customers. Such License Fee shall be payable in accordance with the provisions of Paragraph 4.2 of this Agreement.

4.2 Within twenty (20) days after the end of each calendar quarter ending after the date on which Products are first shipped, LICENSEE shall furnish to Commodore, a written report setting forth the quantity of Products shipped during such quarter and the computation of the aggregate License Fee payable with respect thereto as calculated using the per unit License Fee set forth in Exhibit 1. Each unit shall be accompanied by a payment in the amount of the aggregate License Fees due with respect to that quarter. Commodore shall be entitled to receive interest on any late payments hereunder at a rate of 1 1/2% per month or such lesser rate, if any, as may be required by law.

4.3 LICENSEE shall keep books and records in sufficient detail to permit ready and accurate determination of any License Fees payable by it hereunder. Commodore shall have the right, on one occasion per calendar quarter on not less than seven (7) business days' prior notice, to cause its own accountants or any independent certified public accountant or firm of such accountants to audit such books and records to verify any such determination. Licensee shall provide all reasonable cooperation. Commodore shall pay all costs of such accountants unless such audit determines an unremedied underpayment to Commodore of more than 10% of the aggregate amount properly due to Commodore as revealed by the audit. In such event LICENSEE shall pay all such costs.

4.4 If Commodore shall grant a license with a scope and on terms and conditions substantially equivalent to that set forth herein at a royalty rate more favorable than that provided in this Agreement, then LICENSEE shall be entitled to receive the same favorable royalty rate, subject to the terms and conditions under which such more favorable royalty rate was granted, provided that this paragraph shall not be construed to apply to licenses granted for other than monetary consideration.

5.0 PROPRIETARY RIGHTS PROTECTION

5.1 Each party acknowledges that by virtue of this Agreement, it may gain access to information that is confidential and/or proprietary to the other party and that is so marked in writing, including without limitation, the Product, the Work, the Licensed Software and other trade secrets related to the CDTV Player and the manufacture of Products compatible therewith ("Confidential Information"). For three (3) years from the effective date of this Agreement the party gaining access to (the "Recipient") Confidential Information hereunder, will use all reasonable efforts, but not less than the same degree of care it employs with its own like confidential information, to maintain in confidence all Confidential Information, to avoid disclosing the same to any third party, and to avoid using or permitting others to use the same, commercially or otherwise, in any manner contrary to the purposes of this Agreement or the best interests of the other party.

5.2 Recipient shall only disclose Confidential Information to those of its employees who have a need to know the same in order to accomplish the purposes of this Agreement. Prior to disclosing any Confidential Information hereunder, Recipient shall inform such employees who are to have access to the Confidential Information, of Recipient's limitations, duties and obligations regarding the use, nondisclosure and copying of the Confidential Information and shall obtain the written agreement of such employees to comply with those limitations, duties and obligations. The Recipient shall maintain records of its employees having access to confidential information. Upon reasonable notice to the Recipient the party disclosing Confidential Information may audit such records.

5.3 Notwithstanding the above, Recipient shall have no obligation with respect to any information which: (1) was already known to Recipient at the time of receipt hereunder; (2) is or becomes public or rightfully received by the Recipient from a third party without similar restriction and without breach hereof; (3) is independently developed by the Recipient without benefit of any disclosure hereunder, or (4) is approved for release by written authorization of the disclosing party.

5.4 Upon proper termination of this Agreement by either party, use of all Licensed Software, Commodore's Proprietary Rights and Proprietary Marks by LICENSEE shall be discontinued, and the licenses and rights granted hereunder (except with respect to end-users of the Products shipped prior to such termination) shall expire and LICENSEE shall have no further rights or access to the Licensed Software, Commodore's Proprietary Information and Proprietary Marks. Within thirty (30) days after any such termination of this Agreement all copies of the Licensed Software (except as has been properly included in the Product hereunder) and all other materials containing any copy of any Confidential Information shall be returned to Commodore, or destroyed on Commodore's written instructions. If such termination is not a result of any breach of this Agreement by LICENSEE, LICENSEE may continue to distribute such Products as have been manufactured hereunder prior to such termination providing LICENSEE shall remain obligated to pay the License Fee for each unit of Product shipped.

5.5 LICENSEE shall not disassemble, decompile or otherwise reverse engineer the Licensed Software.

5.6 LICENSEE shall not remove or otherwise obliterate any trademark, patent, copyright or other proprietary rights notices or markings included on or in the Licensed Software, such Product master image as is prepared by the use of the Licensed Software, or any other materials supplied to LICENSEE hereunder, and shall reproduce and apply any of the same in and to any copies of such Licensed Software, Products and materials, in whole or in part, and in any form.

5.7 Notwithstanding any termination of this Agreement, any remaining obligations of the parties as set forth in this Section 5 shall survive this Agreement.

6.0 OTHER DUTIES AND OBLIGATIONS OF LICENSEE

6.1 Within ten (10) days after the initial manufacture of CD-ROM copies of the Product, LICENSEE shall cause to be furnished to Commodore at no additional charge to Commodore, to the attention of CDTV Products Coordinator, c/o Commodore International, 1200 Wilson Drive, West Chester, PA. 19380, or such other address as is specified by Commodore in writing, ten CD-ROM copies of such manufactured hereunder, including such end-user documentation and materials as are to be generally furnished with the Product, provided that as a condition precedent, Commodore enter into LICENSEE's standard end user license agreement covering such copies and materials. Such copies may be retained by Commodore, subject to such license agreement.

6.2 LICENSEE shall use all reasonable efforts to comply with such quality control standards, user interface standards and disc packaging standards as are generally promulgated by Commodore with respect to products for the CDTV Player, provided that such standards are reasonably promulgated by Commodore, are appropriate to the particular product, and are given to LICENSEE in a timely manner. LICENSEE will promptly notify Commodore regarding any aspects of the Product for which LICENSEE will not substantially comply with any such standards that are in effect at the time LICENSEE releases the Product to manufacturing.

6.3 LICENSEE shall ensure that the Product manufactured hereunder is a professional, high quality product that LICENSEE reasonably and in good faith believes will not diminish or otherwise negatively affect the reputation and good will associated with Commodore and the Commodore Proprietary Marks licensed hereunder.

7.0 WARRANTIES

7.1 Commodore hereby warrants that it (a) has the right and power to enter into this Agreement and (b) owns or possesses from others (i) all right, title and interest in and to the Licensed Software, Commodore's Proprietary Rights, Proprietary Marks and CDTV Player, and (ii) the right to grant the rights and licenses set forth herein.

7.2 The Licensed Software and other materials provided by Commodore to LICENSEE hereunder are provided on an "AS IS" basis, without any warranty of any kind except as provided in Paragraph 7.1, above.

7.3 Except with respect to the Licensed Software, Commodore's Proprietary Rights and Proprietary Marks, which are included in the Product (and which are the subject of the Commodore warranty as set forth above), LICENSEE hereby warrants that it possesses sufficient right, title and interest in and to the Product, to reproduce, display and perform such Product, and to authorize others to so reproduce, display and perform the Product as is required by this Agreement. Except with respect to such Licensed Software, Commodore's Proprietary Rights and Proprietary Marks, which are included in the Product LICENSEE shall be solely responsible hereunder for obtaining such right, title and interest in and to the Product, and the continuance of such warranty shall be a condition precedent to such license grants as are made by Commodore herein.

7.4 NEITHER LICENSEE OR COMMODORE MAKE ANY OTHER EXPRESS OR IMPLIED WARRANTIES INCLUDING, BUT NOT LIMITED TO, ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

8.0 LIMITATION OF LIABILITY

8.1 IN NO EVENT WILL EITHER PARTY BE LIABLE FOR ANY LOST REVENUES OR PROFITS, OR OTHER SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, EVEN IF SUCH PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

9.0 TAXES

All payments required under Section 4.0 or otherwise under this Agreement are exclusive of withholding and other taxes, and LICENSEE agrees to bear and be responsible for the payment of all such taxes, including, but not limited to, all sales, use, rental receipt, personal property or other taxes which may be levied or assessed in connection with this Agreement and excluding only such taxes as are based upon Commodore's income.

10.0 MISCELLANEOUS

10.1 All notices and other communications hereunder shall be in writing and effective when personally delivered, or transmitted by telephone facsimile with a confirmation copy by U.S. first class air mail or foreign equivalent, to a party at its address or addresses and facsimile numbers set forth below, or to such other address or addresses as such party shall have last notified to the other. If mailed, such notice shall be deemed received by the close of business on the date shown on the certified or registered mail receipt, or when it is actually received, whichever is sooner.

10.2 If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, then, that provision notwithstanding, this Agreement shall remain in full force and effect and such provision shall be deemed omitted.

10.3 This Agreement and the attached Exhibit 1 which is incorporated herein by reference, constitute and express the final, entire and exclusive agreement and understanding between the parties and supersedes all previous communications, representations or agreements, whether written or oral, with respect to the subject matter hereof.

10.4 This Agreement may not be modified, amended, rescinded, canceled or waived, in whole or in part, except by a written instrument signed by the parties.

10.5 Except in connection with the sale of all or substantially all of LICENSEE's assets, or the sale or transfer of a product line that includes the Product, or the sale of LICENSEE or its business (by merger or otherwise), any of which may be done without the consent of Commodore, this Agreement, including any licenses and rights granted hereunder, may not be sold, leased, assigned, sublicensed (except as specified in paragraph 2.2) or otherwise transferred, in whole or in part, by LICENSEE, without the prior consent of Commodore, which shall not be unreasonably withheld or delayed. This Agreement shall inure to the benefit of and be binding on any third party's successor or any party to whom a transfer hereunder is permitted.

10.6 This Agreement is made under and shall be governed by and construed in accordance with the laws of the Commonwealth of Pennsylvania, excluding its conflicts of law provisions. The parties agree to submit to the exclusive jurisdiction of the appropriate courts located in Pennsylvania for the purpose of any suit, action or other proceeding in connection with this Agreement.

10.7 All payments hereunder shall be made in U.S. Dollars by wire transfer or by check addressed to the payee by U.S. first class air mail (or foreign equivalent) at the payee's applicable address set forth in Exhibit 1.

10.8 In addition to any other relief, the prevailing party in any action arising out of this Agreement shall be entitled to reasonable attorneys' fees and costs.

IN WITNESS WHEREOF, the parties have caused this Agreement to be executed by their duly authorized representatives.

COMMODORE HOLDING B.V.

LICENSEE (print or type only)

By: _____

By: _____

Name: _____

Name: _____

Title: _____

Title: _____

Address: _____

Company: _____

Address: _____

Fax No.: _____

Phone No.: _____

Fax No.: _____

Developer No.: _____

Date: _____

Date: _____

EXHIBIT 1

The Exhibit is attached to and made a part of that Agreement between Commodore Holding B.V. and
("LICENSEE") dated and effective _____, 19____.

LICENSEE: _____

WORK:

Address: _____

Description: _____

UNIT LICENSE FEE: Twenty-five (\$.25) cents

PROPRIETARY MARKS:

CDTV and the CDTV logo

PAYMENTS SHALL BE MADE TO: Commodore Holding B.V., c/o Commodore Business Machines, Inc., P.O. Box
7777-W6300, Philadelphia, PA 19175

COMMODORE HOLDING B.V.

LICENSEE (print or type only)

By: _____

By: _____

Name: _____

Name: _____

Title: _____

Title: _____

Address: _____

Company: _____

Address: _____

Fax No.: _____

Phone No.: _____

Fax No.: _____

Developer No.: _____

Date: _____

Date: _____

RJG/0292:mr

Resources

The companies listed below pre-master, master, and/or replicate CD-ROMs. They should be contacted individually for specialties and pricing information.

These companies have been approved for pre-mastering and creating write-once discs

Next (UK)
Elektroson (Netherlands)
Clarinet (UK)
Discovery Systems (USA)

The other companies listed below claim to have pre-mastering services ready. However, we have not verified their capabilities. Please contact those companies directly for more information on the pre-mastering services they offer, the data formats they accept as input, their experience in CDTV Multimedia pre-mastering, etc.

3M OPTICAL RECORDING DEPARTMENT

3M Center 223-5S-01

St. Paul, MN 55144-1000

Telephone

612-736-5399

Fax

612-733-0158

Sales contact

Don Winklepleck

603-595 0391

Technical contact

Andy Axelsen

800-336 3636

Fax

715-235 0500

Territories served

North and South America, Europe, Far East

Preferred Source format

8mm Exabyte, 4 mm DAT, 9 Track, MO, Hard Disk, CDWO, 3480, DC6150 (QIC)

One-off service available

Call

Pre-mastering and CDTV ISO building

Call

Terms of Business

Net 30 days

ADVANCED MEDIA GROUP, LTD.

P.O. Box 1623

Lancaster, PA 17603

Telephone

717-392-6533

Fax

717-392-0532

Sales contact

Stan Caterbone

Technical contact

Stan Caterbone/Mike Hess

Territories served

North and South America, Europe, Far East

Preferred source format

Tape, hard disk. All media accepted upon prior approval

Terms of Business

Upon invoice Net 30 with prior approval

One-off service

No

Pre-mastering and CDTV ISO building

No

AMERIC DISC INC.

2525 Canadien
Drummondville, Quebec
CANADA J2B8A9

Telephone

Fax

Sales contact

Technical contact

Territories served

Preferred source format

Terms of Business

One-off service

Pre-mastering and CDTV ISO building

819-474-2655

819-474-2870

A. Frank Johansen

Peter Frame

Canada, USA, South America

8mm Exabyte, 9 track tape, CD-ROM

Net 30 days on credit approval

No

No

ATTICA CYBERNETICS LTD.

Unit 2 Kings Meadow
Ferry Hinksey Road
Oxford, England OX2 0DP

Telephone

Fax

Sales contact

Technical contact

Territories served

Preferred source format

Terms of Business

One-off service

Pre-mastering and CDTV ISO building

44-865-791-346

44-865-794-561

Gill Diskson

Ian Ellison

Worldwide, from UK office

AmigaDOS file on SCSI drive. Other
options availablePayment in advance; credit for UK com-
panies by arrangement

Call

Call

CLARINET SYSTEMS LTD.

White Hart House
London Road
Blackwater, Camberley, Surrey, UK

Telephone

Fax

Sales contact

Technical contact

Territories served

Preferred source format

One-off service

Pre-mastering and CDTV ISO building

44-276-600-398

44-276-600-592

Stephen Schoiefield

Chris Simmonds

UK, Europe

Hard disk, DAT, floppy, Exabyte, 1/2 inch tape

Available

Available

DIGIPRESS

2516 River Bend Drive
Louisville, KY 40206

Telephone

502-895-0565

Contact

Dennis Oudard

In Europe, contact

DIGIPRESS

10 rue de Paris
78100 Saint-Germain-en-Laye
France

Telephone

33-1-30-61-11-00

Contact

Marc Deflassieux

DIGITAL AUDIO DISC CORPORATION (DADC)

1800 N. Fruitridge Ave.
Terre Haute, IN 47804

Customer service Telephone

812-462-8192

Customer service Fax

812-466-2007

Sales contact

Bob Hurley

Telephone

603-595-4331

Fax

603-595-4310

Technical Contact

Cliff Brannon

Telephone

812-462-8286

Fax

812-466-9125

Territories served

USA, Europe, Far East

Preferred source format

9 track tape, MO, 8mm, 4mm, CD WO

Terms of Business

1% 10 Net 30

One-off service

Available

Pre-mastering and CDTV ISO building

Available soon

DISCOVERY SYSTEMS

7001 Discovery Boulevard
Dublin, OH 43017

Telephone

614-761-2000

Fax

614-741-4258

Sales Contact

Greg Tiller

Technical Contact

Alex Deak, Customer Service

Territories served

North America, Europe, Australia

Preferred Source format

9 track or 8mm tape. Call for additional options.

One-off service

Available

Pre-mastering and CDTV ISO building

Yes

Terms of business

90 days net

DISC MANUFACTURING, INC.

1120 Cosby Way
Anaheim, CA 92086

Telephone

Fax

Sales contact

Technical contact

You may also contact

DISC MANUFACTURING, INC.

A Quixote Company
4905 Moores Mill Rd.
Huntsville, AL. 35810

Telephone

Fax

Sales contact

Technical contact

Territories served

Preferred Source format

One-off service

Pre-mastering and CDTV ISO building

Terms of Business

714-630-6700

714-630-1025

Wan Seegmiller

Leon Whidbee

205-859-9042

205-859-9932

Kim Vandenberghe

Shogo Karitani

North and South America, Europe, Far East

8mm Exabyte, Pinnacle MO, One-off

CD, 9- track computer tape

Call

Call

Net 30 days, on credit approval

ELEKTROSON

Velderseweg 25
5298 LE Liempde

THE NETHERLANDS

Telephone

Fax

Sales and technical contact

Territories served

One-off service

Pre-mastering and CDTV ISO building

31-4113 3021

31-4113 2763

Dr. R.C.H. BROERS

Europe

Yes

Yes

M.P.O.

195 Ave. Charles de Gaulle
92200 NEUILLY SUR SEINE

FRANCE

Telephone

Fax

Sales contact

Technical contact

Territories served

Preferred Source Format

One-off service available

Pre-mastering and CDTV ISO building

Terms of business

331-4722 2000

331-4722 6077

Bruno d'ORGEVAL

Marc des RIEUX

USA/Canada (Disc Americ)

Europe (MPO), Spain (Techno-CD)

Video 8mm

Yes

Yes

60 days

MULTI MEDIA MASTERS & MACHINERY SA

Av. des Sports 42

CH - 14000 YVERDON-LES-BAINS

SWITZERLAND

Telephone

41-24 23 71 11

Fax

41-24 23 71 12

Sales contact

Gregory KOLER

Technical contact

Albert KHOURY

Territories served

Europe, USA if required

Preferred Source Format

9 track tape, 8 mm Exabyte, 3/4 U-Matic,
DAT, 5.25" Optical

One-off service

Call

Pre-mastering and CDTV ISO building

Call

Terms of business

Net 30 days

NEXT TECHNOLOGY CORPORATION LTD.

St. John's Innovation Center

Cambridge

Cambridgeshire CB4 4WS

UNITED KINGDOM

Telephone

44-223 420 222

Fax

44-223 420 015

Sales contact

Ian Thomas

Technical contact

Neil Critchell

Territories served

Europe

Pre-mastering and CDTV ISO building

Yes

NEXT specializes in pre-mastering and CDTV ISO building services

NIMBUS INFORMATION SYSTEMS

SR 629, Guildford Farm

Ruckersville, VA 22968

Telephone

804-985-1100 or 800-782-0718

Fax

804-985-4625

Sales contact

Larry Boden

Technical contact

Ernest Runyon

In Europe, contact

NIMBUS INFORMATION SYSTEMS

Wyastone Leys, Monmouth

GWENT NP5 3SR

United Kingdom

Telephone

44-600-890682

Fax

44-600-890779

Sales contact

Steve Connolly

Technical Contact

Jim Orr

Territories served

USA and Europe

Preferred source format

4mm DAT or 8mm Exabyte

Terms of business

Net 30 days, upon credit approval

One-off service

Call

Pre-mastering and CDTV ISO building

Call

ON-SITE CD SERVICES

13901 Lyndie Ave.
Saratoga, CA 95070

Telephone

408) 867 0514

Fax

408-867 0518

Sales contact

Rick Wittwer

Technical contact

Lance Buder

Territories served

USA

Preferred source format

8 mm. ANSI labeled image or hard disk

One-off service

Call

Pre-mastering and CDTV ISO building

Call

On-Site is a pre-mastering specialist, located in California. They do not replicate discs in quantity.

OPTICAL MEDIA STORAGE S.P.A.

Località Campo di Pilek
67100 L'Aquila

ITALY

Telephone

39-862 3311

Fax

39-862 315366

Technical contact

Antonio Bruno

Territories served

Europe

Preferred source format

9 track tape, hard disc, CD-WO, CD-ROM ISO image

One-off service

Call

Pre-mastering and CDTV ISO building

No

PHILLIPS AND DU PONT OPTICAL

1409 Foulk Road, Suite 200
Wilmington, DE 19803

Telephone

302-479-2501

Fax

302-479-2512

Sales contact—USA

Joe Bradley

Telephone

301-989-9341

Technical contact

Jim Fricks

Telephone

704-734-4211

Territories served

USA and Europe

Preferred source format

8mm, 9-track tape, one-off CD

Terms of business Net 30 days

One-off service

available in USA

Pre-mastering and CDTV ISO building

USA only

P.D.O. offers pan-European services. Here are contact addresses and numbers in other countries.

PDO Headquarters

Building EF-2

P.O. Box 218

5600 MD Eindhoven

THE NETHERLANDS

Telephone

31-40-751120

Fax

31-40-757866

PDO Technical Support—Europe

Klusriede 26

D-3012 Langenhagen 1

GERMANY

Telephone

49-511-7306-253

Fax

49-511-7306-694

PDO sales office for Benelux and Italy

Building EF-2

P.O. Box 218

5600 MD Eindhoven

THE NETHERLANDS

Telephone

31-40-751120

Fax

31-40-757866

PDO sales office for the UK, Ireland and Scandinavia

Queen Anne Nouse

11 The Green

Richmond upon Thames

Surrey TW9 1PX

UK

Telephone

44-81-948-7368

Fax

44-81-940-7137

PDO sales office for France and Spain

43 Avenue Marceau

F-75116 Paris

FRANCE

Telephone

33-1-4070-1123

Fax 33-1-4070-1126

PDO Sales office for Germany, Austria and Switzerland

Adenauerallee 32

D-2000 Hamburg 1

Germany

Telephone 49-40-280-1391

Fax

49-40-280-1785

SONOPRESS GMBH

Carl-Bertelsmann-Str. 161

D-4830 Gutersloh 100

Germany

Telephone

Fax

Sales contact

Technical contact

Telephone

Fax

Territories served

Preferred source format

Terms of business

One-off service

Pre-mastering service

49-5241-803074

49-5241-73686

Dr. Reinhard Raubenheimer

Ulrich Graznow

49-5241-805250

49-5241-75863

all European countries, USA if required

9 track tape, Exabyte, SCSI harddisk. Au-

dio input media Sony 1610/1630

Based on customer requirements

Call

Call

Sonopress provides pan-European services. Here are contact names and numbers in other countries.

SONOPRESS UK

Sales contact

Telephone

Fax

Sabine Leuerer

44-71-499-6813

44-71-493-7244

SONOPRESS FRANCE

Sales contact Madame Bornhold

Telephone

Fax

33-1-45-636707

33-1-43-596673

SONOPRESS ITALY

Sales contact Dr. Paolo Montagna

Telephone

Fax

39-2-7600-4737

39-2-7601-5026